Towards a Safe and Modular Architecture for Autonomous Drone Autopilots

Matheus Ladeira Boechat Lemos

THESIS

for obtaining the title of

Doctor of Philosophy

Delivered by:
École Nationale Supérieure de Mécanique et d'Aérotechnique (ISAE-ENSMA)

Defended on *14/11/2023* **by:**
Matheus LADEIRA BOECHAT LEMOS

Towards a Safe and Modular Architecture for Autonomous Drone Autopilots

---

JURY

Sanjoy BARUAH	Full Professor	Referee and Examiner
Chokri MRAIDHA	Researcher	Referee and Examiner
Emmanuel GROLLEAU	University Professor	Advisor and Examiner
Yassine OUHAMMOU	Lecturer	Advisor and Examiner
Joël GOOSSENS	Full Professor	Examiner
Liliana CUCU-GROSJEAN	Research Director	Examiner and President of the Jury

---

Doctorate School and Specialization:
MIMME: Mathématiques, Informatique, Matériaux, Mécanique, Energétique
Research Unit:
Laboratoire d'Informatique et d'Automatique pour les Systèmes (LIAS)
Thesis Advisors:
Emmanuel GROLLEAU and *Yassine OUHAMMOU*
Referees:
Sanjoy BARUAH and *Chokri MRAIDHA*

Acknowledgements

I would like to begin by thanking my advisors, Manu and Yassine, for their guidance, the energy and time spent in this project, the valuable knowledge shared with me throughout the last years, and the effort that was required to adapt to the harsh conditions imposed by a global pandemic. Yassine, merci beaucoup pour la patience et la persistance devant ces périodes éprouvantes. Manu, merci beaucoup pour la gentillesse, l'encouragement et la confiance que tu as déposée sur moi. Tout cela a été fondamental non seulement pour la production de cette thèse, mais aussi pour bâtir des bases solides pour ma formation et pour avoir un impact si important dans ma carrière.

Thank you to the referees Sanjoy Baruah and Chokri Mraidha, and examiners Liliana Cucu-Grosjean and Joël Goossens, for having accepted to review this thesis.

I am also very thankful towards the doctorate school MIMME (f.k.a. SISMI), the engineering school ISAE-ENSMA, the research lab LIAS and the European project COMP4DRONES for providing me with all the physical and administrative resources I needed in order to accomplish my research. Also, I am grateful for the help of several COMP4DRONES partners, without which this development would be much harder. In addition, thank you to the Realtime-at-Work staff, who allowed me to finish writing this thesis in the best possible conditions.

I am especially grateful for the doctorate colleagues I had, with whom I was lucky enough to be able to share the burdens and joys of my path. Ali, Jorge, Richard, Louise, Maxime, and many others, thank you for making me feel I was not alone. Gracias especialmente a Jorge, Gad, Marcos y Melissa, quienes me recibieron con los brazos abiertos y me trajeron la luz y los colores de nuestra latinoamérica aunque estemos tan lejos de casa.

A huge thanks to my lab colleagues for making the everyday life much nicer. Un merci spécial à Bénédicte et Mickael : vous avez été essentiels pour que je puisse résoudre toute sorte de problème et, sans vous, je ne serais sûrement pas arrivé jusqu'ici. Likewise, thank you very much to my interns, Carlos and Yuri, who have helped me advance my research significantly.

Aos meus compatriotas expatriados, meus amigos brasileiros Luciana, Babi, Ramon, Fábio e todos os outros que tive o prazer de encontrar, obrigado por me fazerem me sentir em casa. Obrigado pelas festas, pelas conversas, pelas sessões de descarrego, pelas dicas de eventos, pelo socorro administrativo nos labirintos burocráticos desta república estrangeira. Sem vocês, teria sido impossível.

A minha família de sangue – meu pai, minha mãe, meu irmão, minha avó, tios e primos –, obrigado por acreditarem em mim, por me mostrarem o valor do conhecimento, pelo suporte inigualável e pelo encorajamento que me fazem ter a certeza de que fui muito privilegiado de ter nascido onde nasci. E à minha família de coração, Marina (prima!) e Thomas, obrigado

pelo suporte emocional, pelas conversas sem fim, pelos jogos de tabuleiro, pelos apéros, pela companhia nos momentos mais altos e mais baixos, por tornarem o confinamento tão mais leve e divertido, e por trazerem tanta cor, sabor, som e calor à minha vida. Amo muito todos vocês!

Não posso deixar de agradecer a todos os professores e profissionais da educação que tanto me marcaram: Alexandre, Amanda, Daniel, Débora, Donizete, Erick, Gabriel, Helenice, Lorena, Marcelo, Nilvane, Renato (cheers!), Sander... Muitíssimo obrigado não só pelo conhecimento transmitido, mas pelo raciocínio crítico, o encorajamento e a paixão pelo conhecimento compartilhados comigo desde cedo. Vocês inspiraram a mim e a tantos outros. Obrigado pelo ofício tão importante. Em especial, obrigado Erick pelo tempo e esforço cedidos para me ajudar a reduzir o peso desse longo e árduo percurso.

Finally, to all those who believe in a public educational system, in an open science environment, and in an international cooperation to improve the quality of life for all humans regardless of any differences, it is thanks to you that I was able to arrive where I am today. Thank you with all my heart.

Contents

List of Figures

List of Tables

Table of Acronyms

AADL Architecture Analysis and Design Language.
ADL Architecture Description Language.
AHRS Attitude and Heading Reference System.
APEX Application Executive.
API Application Programming Interface.
ASIL Automotive Safety Integrity Level.
AUTOSAR Automotive Open System Architecture.

COTS Component Off-The-Shelf.

DAG Directed Acyclic Graph.
DAL Design Assurance Level.
DDS Data Distribution Service.
DDSI-RTPS Data Distribution Service Interoperability Wire Protocol - Real-Time Publish-Subscribe.
DSL Domain Specific Language.
DSML Domain Specific Modelling Language.

EASA European Union Aviation Safety Agency.
EDF Earliest Deadline First.
ENAC *École Nationale d'Aviation Civile*.

FAA Federal Aviation Administration.
FIFO First-In-First-Out.
FMU Flight Management Unit.

GCD Greatest Common Divisor.
GCRT Generalized Chinese Remainder Theorem.
GCS Ground Control Station.
GNC Guidance, Navigation and Control.
GNSS Global Navigation Satellite System.
GPOS General Purpose Operating System.
GPS Global Positioning System.

HAL Hardware Abstraction Layer.

IMA Integrated Modular Avionics.
IMU Inertial Measurement Unit.
INS Inertial Navigation Systems.
IPC Inter-Process Communication.

LCM Least Common Multiple.

MARTE Modeling and Analysis of Real-Time and Embedded Systems.
MAV Micro Air Vehicle.
MAVLink Micro Air Vehicle Link.
MBSE Model-Based System Engineering.
MDE Model-Driven Engineering.
MISRA Motor Industry Software Reliability Association.
MOTS Modifiable Off-The-Shelf.

OMG Object Management Group.
OMT Optimization Modulo Theories.
OS Operating System.
OSATE Open Source AADL Tool Environment.
OSI Open Systems Interconnection.

QoS Quality of Service.

RoBMEX ROS-Based Modelling Framework for End-Users and Experts.
ROS Robot Operating System.
ROS2 Robot Operating System 2.
ROSMiLan ROS Mission Language.
ROSModL ROS Modeling Language.
ROSProML ROS Process Modelling Language.
RTOS Real-Time Operating System.

S&F Store-and-Forward.
SAE Society of Automotive Engineers.
SAIL Specific Assurance and Integrity Level.
SI Simultaneous Incongruences.
SMT Satisfiability Modulo Theories.
SORA Specific Operations Risk Assessment.
SPC Semaphore Precedence Constraints.

TSN Time Sensitive Network.

UAS Unmanned Aerial System.
UAV Unmanned Aerial Vehicle.
UC Use Cases.
UML Unified Modeling Language.

VLOS Visual Line of Sight.
VTOL Vertical Take-Off and Landing.

WCET Worst Case Execution Time.
WP Work Package.

Introduction

Chapter 1

Introduction

1.1 Context

In the last decades, the world has experienced a very sharp increase in civil drone utilization. What was once confined to the military and research domains is now available for several civilians. Although it seems that drones are a novelty, unmanned vehicles are not truly new.

If a vehicle is defined as an object capable of transporting people or other objects to different locations, unmanned aerial vehicles exist at least since the middle of the 19^{th} century: in 1849, Austrians tried to bomb Venice using pilotless balloons with timed bombs [Ale18]. Remotely controllable vehicles, however, appeared some decades later: a patent for the remote control of vehicles exists since 1898 [Tes98], and pilotless aeroplanes were used in military operations in World War I and II [Ale18].

Since then, drones were mostly used for military operations, since not only they avoid putting pilots at risk, but they also remove the most fragile component of an aircraft: the human being. On the other hand, until recent years, they were too big and expensive to be attractive for most civil applications. The complexity of the system required to keep a pilotless vehicle in the air and precisely control its movement called for several heavy embedded equipments, a big obstacle for implementing the technology in quotidian life.

Yet, they have always had the potential to be important tools in various domains [MB20]: monitoring and inspecting large networks, such as pipelines, energy transmission lines, roads and railways; aid the operation of agricultural industries, mines and construction industries; activities of public interest such as surveillance, damage assessments after disasters, fire control, lifesaving in water bodies, humanitarian reliefs and security; logistics such as in managing inventories, transporting medical supplies (e.g. biological samples and organs), applying pesticides, delivering food and other consumer goods; and leisure activities such as photography and racing.

It was only in the past couple of decades that drones became a widespread subject of research and development. This can partially be explained by the electronic components, which have been evolving and achieving smaller proportions and larger efficiency and processing power [Eti18]. In addition, batteries have reached high levels of energy density (joules per gram), making it possible for portable devices to become smaller, lighter and more powerful, as well as more affordable. Drone technology started to become more accessible by the

public.

Their wide range of applications and the new available technologies have proven to create a large interest in developing drones. According to the Teal Group [Gro17][Gro19b], drone system production alone represented a $2.8 billion global annual market in 2017 and $5 billion in 2019, and these values are expected to almost triple to $14.5 billion by 2028, as seen in Figure 1.1 and Figure 1.2. The compound annual growth rate, in constant dollars, is therefore of 12.5%, and drone production is expected to sum up to $97.6 billion from 2019 to 2028. In addition, considering also the services provided by drones, the drone-related economy represents significantly higher values [Via21].

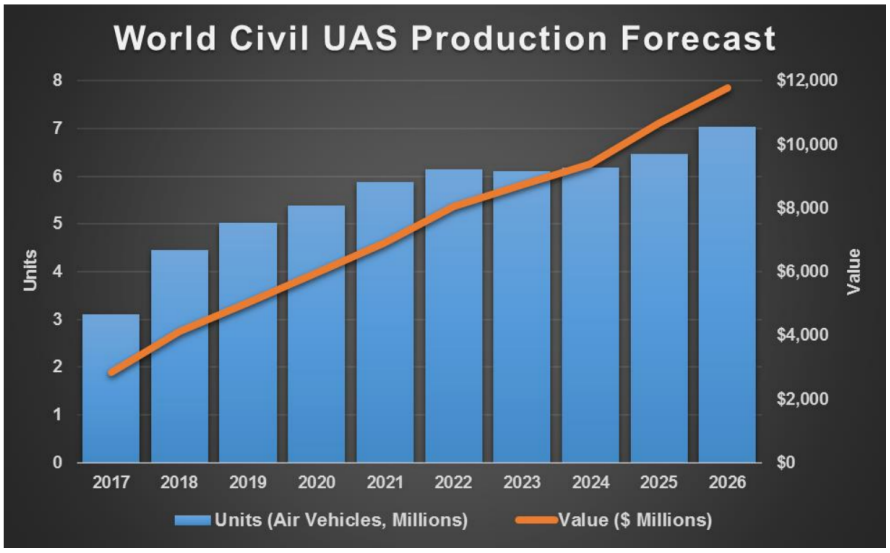

Figure 1.1: Predictions of the global market of UAS from 2017 to 2026 [Gro17]

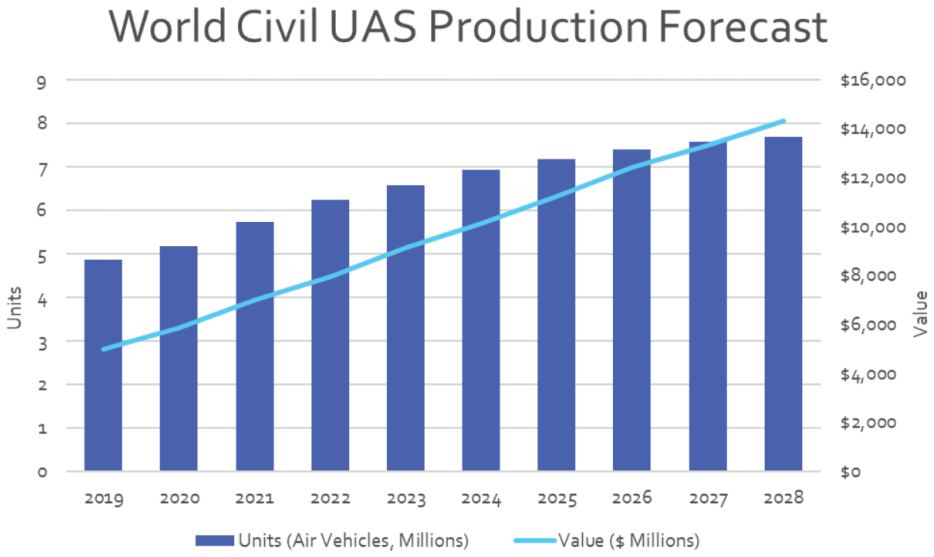

Figure 1.2: Predictions of the global market of UAS from 2019 to 2028 [Gro19b]

Even with such diverse applications and a considerable interest from society, there are still important obstacles to implementing drone utilization. These obstacles are related to the intrinsic complexity of drone systems and the way it has been managed. In addition to understanding the dynamics and aerodynamics of the vehicle, the designers must implement control laws in a real-time embedded system capable of reading and interpreting several sensors, calculate the necessary actions and then execute them in each individual actuator. Therefore, it is impractical to redevelop a whole system from scratch, and reuse is often preferred.

Some open-source drone projects have been managing the complexity of drone embedded systems since the 2000s, notably: ArduPilot[1], PX4[2] and Paparazzi[3]. In addition to providing hardware components, these projects also provide generators for the embedded software that controls the drones – also called *autopilot*. Open-source autopilots have the advantage of being tested and reviewed by several people around the world, in distinct domains of application, in addition to being free. Thus, most drones are developed using these autopilots – or variations of them – as the embedded software.

Using an autopilot as a Component Off-The-Shelf (COTS) raises, however, an important issue. Since the autopilot code is complex, users either consider it a black box and limit themselves to the "factory settings" (not always suitable for the desired application), or they need to acquire a considerable amount of knowledge about the whole system in order to successfully apply any change. Also, the idea of autonomous vehicles sharing space with people and infrastructure creates concerns about what can be done to avoid accidents. Nowadays, regulatory organizations try to define limits to be respected in order to guarantee a sufficient level of safety, but none of them specify details about the embedded software.

In this context, a group of about 50 European organizations, including industrials and academia, has decided to collaborate in order to provide a more complete framework for drone developers. The project, called COMP4DRONES (Components for Drones), aims at making this framework able to reduce the efforts of reconfiguring the drones for specific needs, helping the reuse of external components without necessarily considering each component as a black box – instead, they seek using software pieces as Modifiable Off-The-Shelfs (MOTSs).

In addition, COMP4DRONES seeks providing safety analysis so that not only the physical components can be validated, but also the embedded software. In such manner, it would become easier to ensure the good functioning of the drones and, therefore, their use in civil space.

The work presented here was developed as part of the COMP4DRONES project, as the result of the analysis of drone embedded systems and their improvement in the aspects of modularization and safety of execution.

[1]https://ardupilot.org/
[2]https://px4.io/
[3]https://wiki.paparazziuav.org/

1.2 Definitions

Several communities have been developing their own solutions for drone systems, each one focused on its specific goals and adopting different approaches from different perspectives. Due to this diversity of methods, there has also been significant variations of nomenclatures to refer to the same modules. Therefore, the following definitions will be the reference for this work.

Drone

Definition 1.1. A drone is a remotely controllable or autonomous unmanned vehicle, regardless if it travels on land, in the air or in the water.

A drone is said to be "controllable" when it has mechanisms that are able to change its linear and angular movement at command. The linear movement is related to its position (such as in an xyz reference frame) and the respective rates of change. The angular movement is related to its angular position (its orientation in space, as it is not a point but a three-dimensional body) and the respective rates of change.

Attitude

Definition 1.2. The attitude of a drone is its angular position. It may be described using Euler angles, quaternions, or another similar system.

For example, Figure 1.3 represents transformations that can be a base for Euler angles α, β and γ, which can be used to describe the attitude of the drone. A rotation of α around the z axis, then β around the rotated x axis, and finally γ around the rotated z axis, provides the position of the system $x'y'z'$ with respect to xyz.

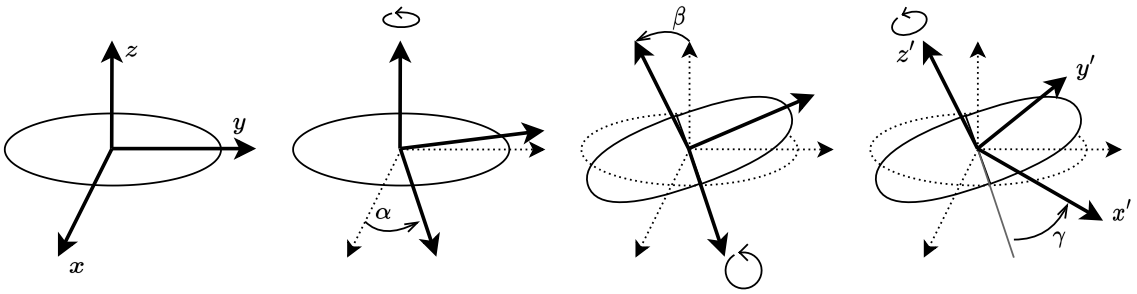

Figure 1.3: Representation of rotations involved in transforming a reference system into another using Euler angles (ZXZ convention)

Therefore, air balloons carrying explosives [Ale18] are not encompassed by the definition used in this thesis of a drone, since their movement cannot be controlled neither by a remote pilot nor does it autonomously control itself. And, although drones share similarities regardless of the medium in which they travel, airborne drones generally need to sustain and stabilize themselves using much quicker movements and reactions, due to the nature of

heavier-than-air vehicle flights. Therefore, the implementation of their control system is often more challenging.

UAV

Definition 1.3. An Unmanned Aerial Vehicle (UAV) is a drone capable of flying.

UAVs can be divided into three main categories:

Fixed-wings: The lift is provided by the movement of the drone itself, comparably to commercial aeroplanes.

Rotary-wings: Lift is provided by rotary blades, and the drone can be static with respect to the atmosphere.

Other types: Vertical Take-Off and Landing (VTOL), lighter-than-air, etc.

Figure 1.4 is a screenshot from QGroundControl[4] used to select the model for the dynamics of the drone. It represents several types of UAVs, and even a kind of rover. Rotary wings are represented as blue and green circles, where each colour represents either a clockwise or an anti-clockwise rotation of the blades.

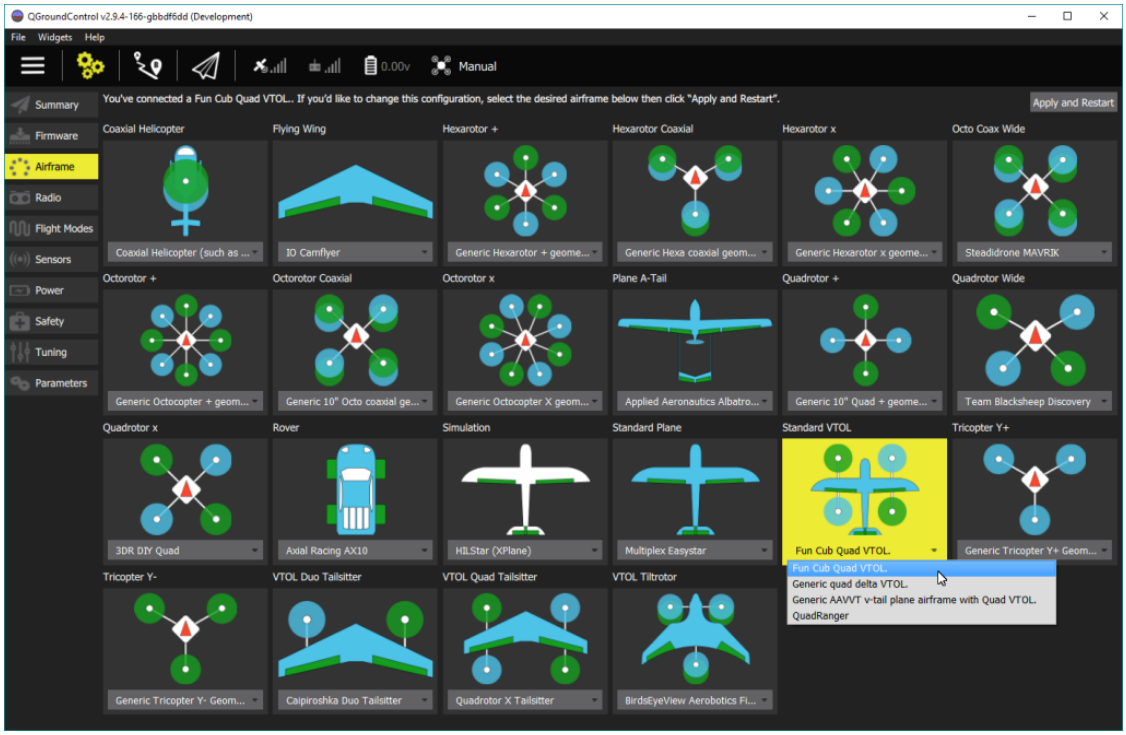

Figure 1.4: Examples of possible drone configurations considered by QGoundControl

Generally, one can interact with a UAV by interfaces such as a remote control or a computer with a communication link with the drone. The computer assumes the role of a GCS.

[4] `https://docs.px4.io/v1.9.0/en/getting_started/frame_selection.html`

UAS

Definition 1.4. An Unmanned Aerial System (UAS) is the system composed of one or more UAVs and one or more GCSs. The GCSs communicate with the UAVs using a message protocol, which allows the transmission of commands and telemetry. The UAVs might also be able to directly communicate between each other.

Autopilot

Definition 1.5. The autopilot of a drone is the embedded software responsible for the functions that measure, plan and control the movement of the drone. It envelops the Guidance, Navigation and Control (GNC) functions.

Other names of the autopilot include *flight stack* and *Flight Management Unit (FMU)*. The autopilot will determine what are the skills and functions of the drone. Depending on the available functions and skills, varying degrees of autonomy will be assigned to the drone, as shown in Figure 1.5 (where RUAV refers to a Rotary UAV and RUAS is a UAS where the vehicles are RUAVs). The following definitions are in line with [Ken12a].

Navigation

Definition 1.6. Navigation is the module responsible for calculating the state vector of the drone (linear and angular positions and their derivatives) using acquired sensor data.

In the Navigation module, we often find filters for sensor data, such as a Kalman filter [WB+95] – responsible for conciliating measurements from different sensors for the same physical quantity, e.g. the position measured by a Global Navigation Satellite System (GNSS) (such as the Global Positioning System (GPS)) and by integrating inertial sensors. Some Navigation modules are referred to as "state estimation" (e.g., in [MKK16]) or "Inertial Navigation Systems (INS)" (e.g., Paparazzi UAV[5]). Also, we will define *Perception* as the ability to perceive objects in the surroundings, which can be part of an extended Navigation module.

Control

Definition 1.7. Control is the module responsible for managing drone motors and actuators in order to stabilize and move the drone to the desired state vector. It compares the current state (provided by the Navigation) to the target state and, using control laws, calculates the necessary forces and torques to be applied. Then, it calculates the individual forces and torques for each actuator, taking into account the dynamics of the vehicle.

The Control module effectively implements the control laws and mechanisms calculated for the specific drone. The most used control strategy is the Proportional-Integral-Derivative

[5] `https://wiki.paparazziuav.org/wiki/Subsystem/ins`

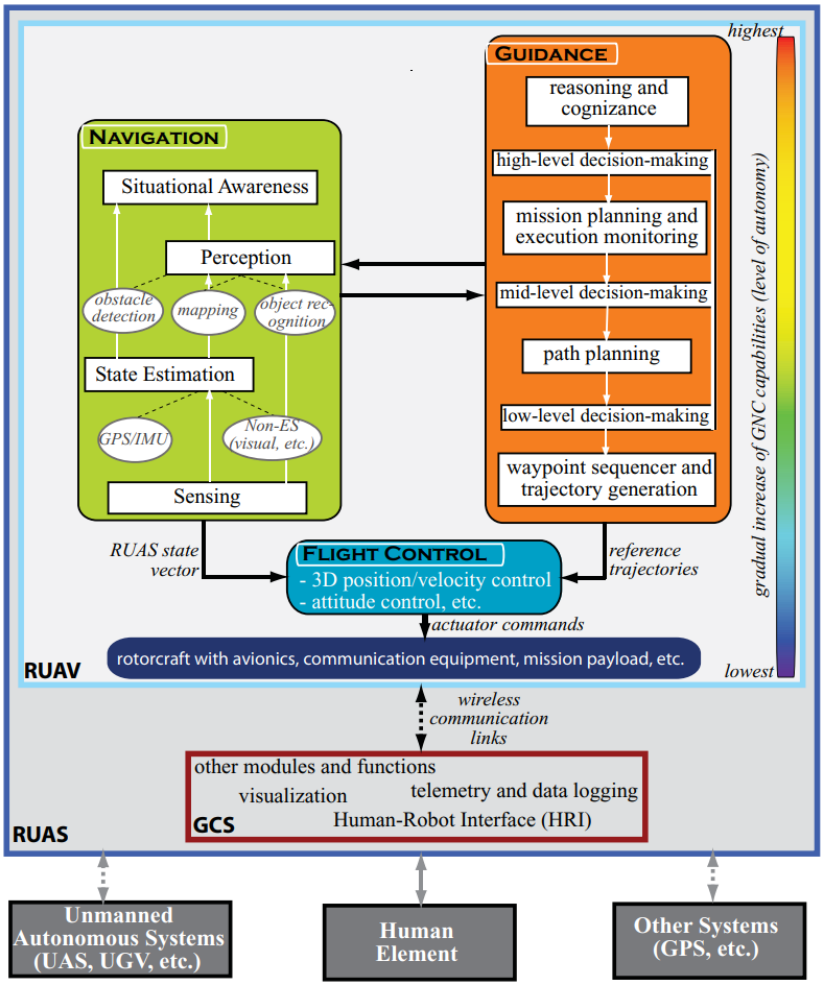

Figure 1.5: Decomposition of a drone autopilot [Ken12a]

(PID), but other strategies can also be used, such as Incremental Non-linear Dynamic Inversion (INDI) [SCM10].

Guidance

Definition 1.8. Guidance is the module responsible for managing the drone's trajectory, using data obtained from the Navigation module and providing target values to the Control module.

The Guidance module envelops the most high-level functions of the drones. The more of its functions are embedded in the autopilot algorithm, the more autonomous the drone tends to be. Trajectory planning and execution can sometimes be called "navigation" (e.g., in [Nik+03] and in Paparazzi[6], which might create confusion.

Autonomy

Definition 1.9. Autonomy is, in this work, approached as the amount of functions that pilots and other humans would have to perform in a remotely piloted drone, but that were embedded in the UAS and, therefore, do not require a human element any more. It is related to the human-independence of a drone.

The least autonomous drones contain only basic stabilization functions in a Control module, are incapable of any Navigation and, therefore, incapable of autonomously executing any Guidance functions: these have to be remotely piloted at all times. A slightly higher level of autonomy is reached when the drone is able to measure its attitude and position, and stabilize them, giving rise to skills such as keeping its position without continuous input from a human pilot. Higher levels of autonomy will include, respectively: trajectory managing, trajectory planning, mission planning, etc.

As it can be seen, drones need to execute several distinct functions at the same time. Each function, implemented in its respective module inside the autopilot, might need to receive and send information to other functions in the system. Albeit the communication can be explicitly managed inside the code of each module, this approach would require that information about the other modules be embedded in its code, which goes against the ideas of modularity. Therefore, it is common practice to use middlewares to deal with internal communication.

Middleware

Definition 1.10. The middleware is a software interface that manages the communication between the modules of a program, so that each module is agnostic regarding other modules.

[6] `https://wiki.paparazziuav.org/wiki/Advanced_Navigation_Routines`

1.3 Autopilots

When conceiving an autopilot, several aspects need to be taken into account. The vehicle dynamics, the on-board hardware, the specific functions and the external communication play a fundamental role in determining the algorithm to be used. Each of these factors has numerous possibilities of choice, creating a combinatorial explosion of configurations.

To deal with the combinatorial explosion, instead of developing specific autopilots for each specific application, the current approach consists in the employment of frameworks that generate autopilots according to the specificities of each application. The autopilot code does not exist independently, since the frameworks generate compiled files directly, and the framework code is considerably complex in order to treat every possible case. This encourages the utilization of the generated autopilots as black-boxes. On the other hand, shedding light into the black-box is fundamental if we envisage overcoming obstacles to mass drone utilization.

Among several available open-source autopilot frameworks, three stand out: ArduPilot[7], PX4[8], and Paparazzi[9]. They are currently maintained, and all of them support autonomous modes. Other autopilots, such as LibrePilot[10], dRonin, CleanFligth, BetaFlight and iNAV are either discontinued (no updates for more than two years) or only have support for remote control and, at most, predetermined flight plans, lacking modes for autonomous flight. Figure 1.6 represents the development timeline of most open-source autopilot frameworks. Dotted lines represent projects which have been discontinued.

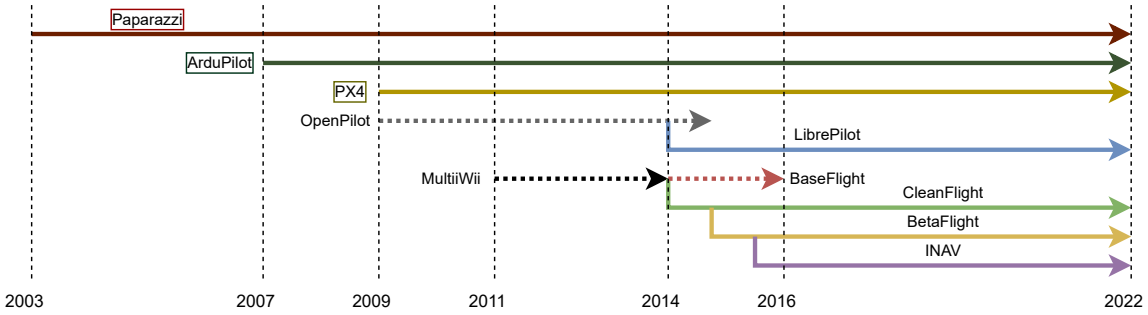

Figure 1.6: Timeline of open-source autopilot frameworks

Although they were developed by different groups in different contexts (and each framework is capable of assessing many different applications), they all share various similarities, as it will be seen in the following subsections.

[7] https://ardupilot.org/
[8] https://px4.io/
[9] https://wiki.paparazziuav.org/
[10] Although LibrePilot presents a "PathPlanner" mode, the trajectory has to

1.3.1 ArduPilot

ArduPilot was born from the founders of 3D Robotics in 2009, having its code publicly available a few months after the release of the first ArduPilot Board. In the beginning of 2010, the ArduPilot code was rewritten from scratch. The 2010 version included "interrupt driven RC [Radio Control] input, RC throttle failsafe, RTL [Return To Launch], Loiter, Circle, Crosstrack correction, decent stabilization, Fly-By-Wire [assisted plane flight], system events, 4 channel RC output, and 2-way telemetry."[11] Then, in that same year, it received support for using Inertial Measurement Unit (IMU) sensors. A Hardware Abstraction Layer was added in 2012, simplifying adding support for different hardware. Nowadays, it is also compatible with the operating system ChibiOS.

Its software architecture is based on a single execution task, a *main loop* that sequentially calls the functions responsible for the autopilot's roles – possibly in addition to tasks responsible for acquiring sensor data and messages from communication buses, in the case of a multitask architecture. Each function has a statically defined period. The main loop, represented in Figure 1.7 and better explained in the contributions of this thesis, is divided into two parts:

Fast loop: The first part to be executed, containing the most critical functions – those related to the movement and stabilization of the drone.

Scheduler: The second part to be executed, containing the least critical functions, is a system-level scheduler. It calls the remaining functions in a best-effort manner, according to priority values statically defined by the coders.

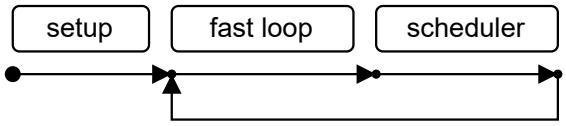

Figure 1.7: Diagram representing the main loop in ArduPilot

In Pixhawk boards, the main loop is executed at 400 Hz, and at 100 Hz in APM 2.x boards. For each board it is deployed to, ArduPilot has a different execution structure, in order to consider the specific implementation characteristics of each piece of hardware. Nevertheless, the distinction in priority levels between critical functions and complementary ones remains the same.

The "fast loop" is executed at the maximum possible frequency, and every function inside it is at the highest priority: their execution is almost guaranteed. The "scheduler", on the other hand, is called after the execution of the "fast loop", and the time for the next iteration of the loop is calculated. For example, if the loop is executed at 100 Hz, the main loop needs to execute at most for 10 ms – otherwise, the frequency will be lower than the configured value. Hence, the scheduler will evaluate how much time has passed since the execution of the current iteration, and subtract it from the total 10 ms available for the execution. If

[11] https://ardupilot.org/dev/docs/common-history-of-ardupilot.html

the value is superior to the expected execution time of its functions, then they are executed. Otherwise, they will be skipped.

ArduPilot provides different flying modes: from simple attitude stabilization for Radio Control, to an autonomous flight given an input mission (sequence of high-level commands). For each mode, a slightly different set of functions is called, but a base structure is common to all modes. It consists in, first, interpreting navigation data, then updating the path (for modes with Guidance enabled), then running the Position stabilization control algorithm (for modes with Position stabilization enabled), then running the Attitude stabilization control algorithm. This cascade control is represented in Figure 1.8, where x refers to the position of the drone, θ is its attitude vector, and ω is its angular velocity – each variable represents a vector, the E subscript indicates the estimated (measured) value, the S subscript indicates a target value (a setpoint), and the dot notation indicates a time derivative ($\omega = \dot{\theta} = \partial\theta/\partial t$).

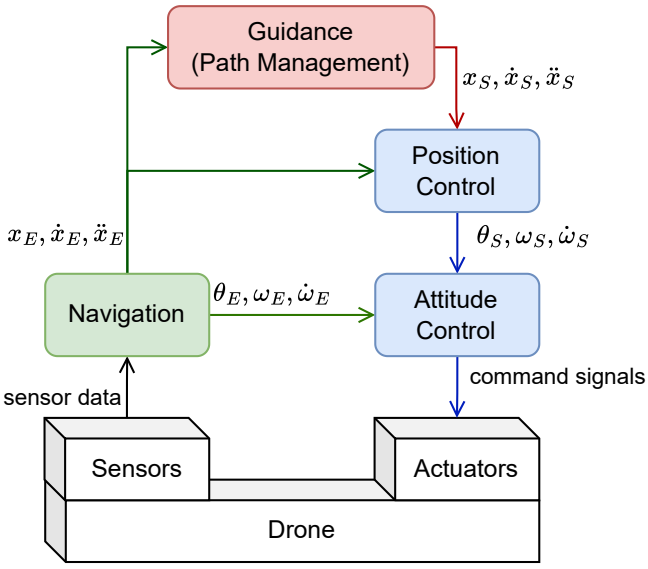

Figure 1.8: Cascade control in a drone autopilot

1.3.2 PX4

PX4 was born from a project aiming to provide a powerful board for drones [Aut]. The project started in 2009, and its goal was to make a modular and easily customizable board, suited for most applications and using easily available components from the market. The team was called Pixhawk, and so was the hardware they developed. Pixhawk was capable of performing 32 bit operations instead of the 8-bit processors frequently used at the time.

In 2012, the 4th version of the Pixhawk software was released, and it was called PX4. As well as the Pixhawk board, PX4 was developed with modularity and versatility in mind. Since the beginning, it was built on top of a POSIX-based real-time operating system: NuttX. Its

modular architecture, presented in Figure 1.9 [Doc][12], eases the understanding and, therefore, the improvement of the software.

From the figure, it is possible to identify a control structure similar to the one in ArduPilot: a cascade control where the Guidance Module output is the Position Control setpoint, and its output is the setpoint of the Attitude Control module (Figure 1.8). In contrast to ArduPilot, the Control architecture is divided into two parallel tasks: a high-frequency task responsible for the Attitude Control, and a low-frequency task responsible for the Position Control and Guidance functions. The communication between modules is managed by a publish-subscribe middleware, uORB[13], which ensures a safe communication between threads (i.e., it is *thread-safe*). PX4 also has modules that bridge uORB and exterior systems, such as ROS and Data Distribution Service Interoperability Wire Protocol - Real-Time Publish-Subscribe (DDSI-RTPS) / Data Distribution Service (DDS). Figure 1.10 [Doc][14] shows the architecture of the bridge connecting PX4 to a DDSI-RTPS system.

1.3.3 Paparazzi

Developed in France by *École Nationale d'Aviation Civile* (ENAC), Paparazzi was born in 2003 as the system responsible for flying a miniature UAV, also called Micro Air Vehicle (MAV) [BH08]. In 2004, the system became able to handle different types of aircraft, and in 2005 it was possible to control a UAS containing multiple UAVs.

Such as in ArduPilot (Figure 1.7), Paparazzi's modules are called sequentially in an eternal loop, starting from the most critical ones – Attitude and Heading Reference System (AHRS), IMU, INS and GPS – and ending with the non-critical ones. Each module is called periodically, with statically defined periods.

The communication between each module is managed by ABI[15], an ad-hoc middleware built on a blackboard-based publish-subscribe pattern. Although Paparazzi is compatible with ChibiOS, its middleware is not thread-safe.

1.3.4 Comparative table

In order to compare these three autopilots, Table 1.1 presents some of their key properties. Cascaded control, modular functions and different flight modes are found in each autopilot, even though they might present small differences. PX4 is the only autopilot to implement parallel tasks for the autopilot's control functions. The others were developed firstly using a bare-metal approach (where no operating system was present). Nowadays, they can implement some functions in parallel tasks using an operating system, but their autopilot functions

[12] `https://docs.px4.io/main/en/concept/architecture.html`
[13] `https://docs.px4.io/main/en/middleware/uorb.html`
[14] `https://docs.px4.io/main/en/middleware/micrortps.html`
[15] `https://wiki.paparazziuav.org/wiki/ABI`

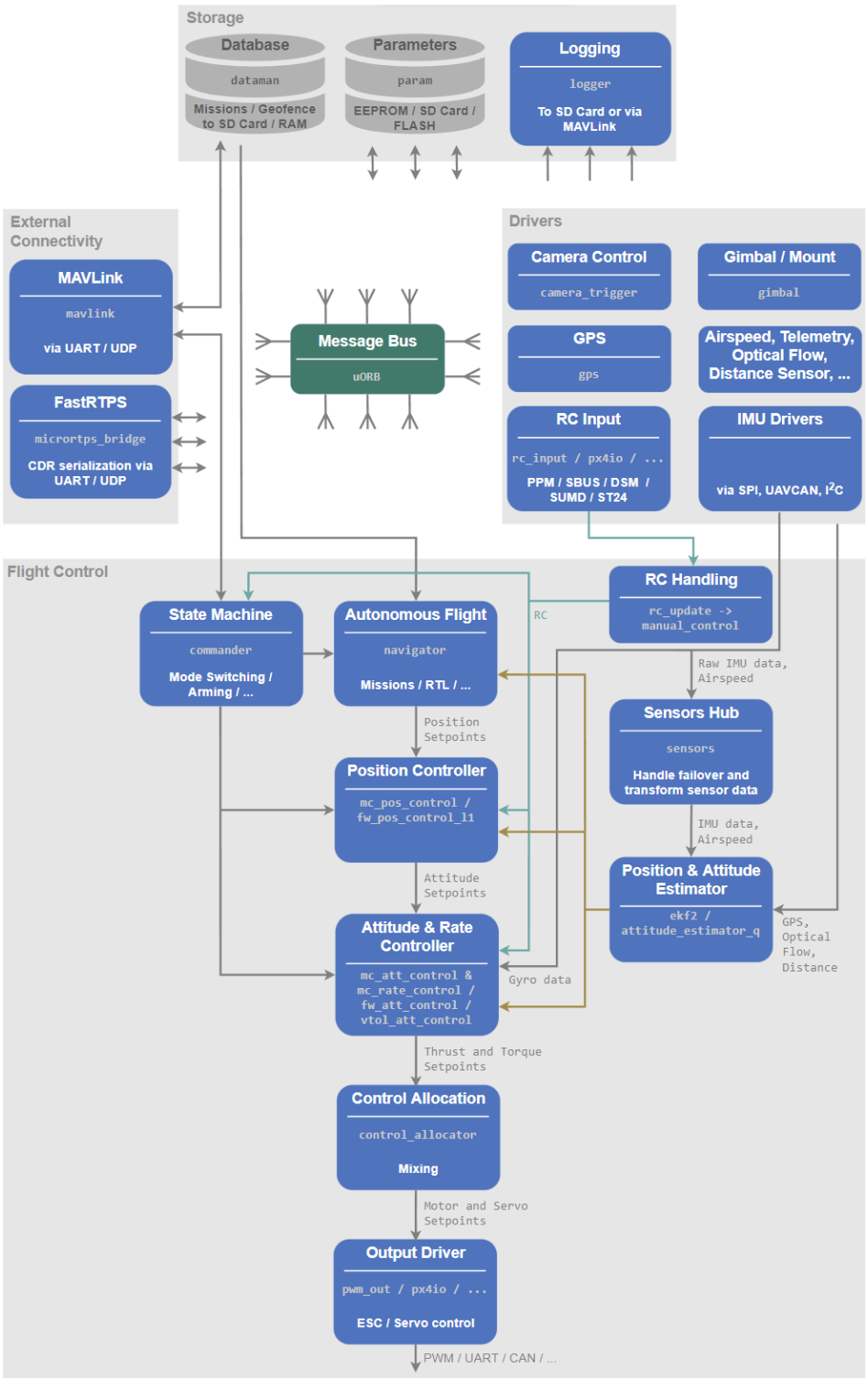

Figure 1.9: Software architecture of PX4 [Doc]

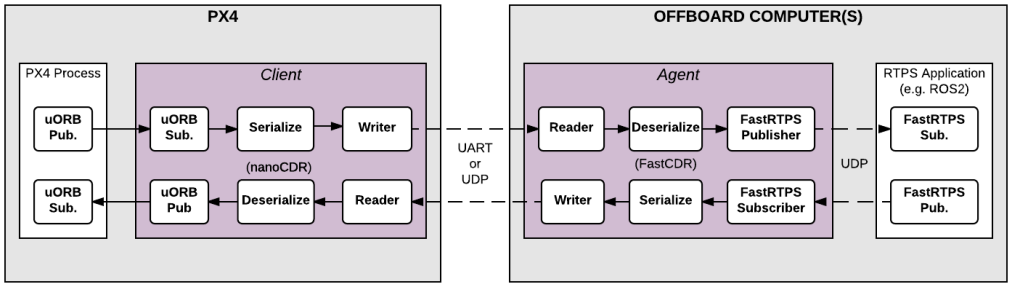

Figure 1.10: PX4 bridge architecture

are still in a single task (or two in the case of PX4). They also keep backwards compatibility, still allowing the bare-metal approach to be used.

While PX4 and Paparazzi use some form of middleware, ArduPilot does not. The communication between the drone and a GCS is done using the MAVLink protocol in the case of PX4 and ArduPilot (nowadays a de-facto standard), and PPRZLink for Paparazzi (developped before the de-facto standard). PX4 also offers native interoperability with ROS and RTPS (DDS). Its licence, despite being open-source, allows companies to sell modifications made to the code, while ArduPilot and Paparazzi are under GPL licence, which forces any derivation of the respective codes to be under the same open-source licence.

Table 1.1: Comparison between open source autopilots for drones

Autopilot	**PX4**	**ArduPilot**	**Paparazzi**
Cascade Control	Yes	Yes	Yes
Modular functions	Yes	Yes	Yes
Flight modes	Yes	Yes	Yes
RTOS / Parallelism	NuttX	- / ChibiOS	- / ChibiOS
Middleware	uORB	-	ABI
Thread-safe Middleware	Yes	No	No
Communication Protocol	MAVLink	MAVLink	PPRZLink
Interoperability	ROS, RTPS (DDS)	-	-
Licence	BSD	GPL	GPL

The autopilots' flight modes can also present some similarities, as seen in Table 1.2. An "attitude rate stabilization" mode refers to the lowest-level loop, responsible for keeping the attitude rates (angular velocities, or ω) at the RC-commanded values. With no input from the RC controller sticks, this mode will automatically reduce ω to 0, keeping therefore the current attitude – if the attitude is not zero, the drone will keep accelerating to one direction. The "attitude stabilization" mode, however, refers to controlling and stabilizing the attitude itself (θ), levelling the drone's roll and pitch. The "position stabilization" mode is capable of maintaining the desired position (x), compensating for disturbances such as wind. "Guidance" mode refers to keeping track of a set of waypoints ($\{x\}$).

Table 1.2: Autopilot modes and relative autonomy level

Modes	**PX4**	**ArduPilot**	**Paparazzi**
ω	Acro	Acro	AP_MODE_RATE_DIRECT
θ	Manual/Stabilized	Stabilize	AP_MODE_ATTITUDE_DIRECT
x	Position, Hold	PosHold, Loiter	AP_MODE_HOVER_DIRECT
$\{x\}$	Mission	Auto	AP_MODE_NAV

1.4 Safety Standards

Guaranteeing that automatic systems will work as intended is not a trivial task when it comes to embedded systems. Not only the hardware has to ensure a certain level of confidence (low level of errors per hour), but the software it carries should be safe – for example, with an absence of bugs, and conforming to specifications and intended functionality. Guaranteeing software safeness in such complex systems and source codes with thousands of lines is far from simple.

In addition, in the domain of vehicles, preventing malfunctions also means preventing important accidents. System bugs might put not only the people inside the vehicle in danger, but also risk people and property around the vehicle. Probably the most known case of embedded systems generating an accident is the case presented in [Koo14]. On the 29^{th} of August of 2009, an accident in the United States involving a Lexus ES 350 killed 4 people and triggered an escalation of investigations: the car had unintendedly accelerated until over 160 km/h. Several similar reports had already been made, and the investigations needed to count on software experts in order to arrive to any conclusion. The code used in the car's equipment was found to be prone to bugs, and the company ended up loosing a class action filed against them.

If the code had complied to known standards and practices, the problems with the embedded systems probably would have been avoided [Koo18]. Therefore, in order to ensure the safeness of UASs, these standards have to be taken into consideration. The most relevant standards for embedded systems are presented in the following sections.

1.4.1 JARUS

The Joint Authorities for Rulemaking of Unmanned Systems (JARUS) are a group of regulation experts from several parts of the world that seek "to develop technical and operational requirements for the safe, secure and efficient operation of UAS, to serve as a common reference for use in respective JARUS Member regulations and guidance, doing it in an effective and efficient manner, avoiding duplication of efforts with other international aviation organizations".[16] In March 2020, 63 countries contributed to the development of JARUS, there being at least one country from every continent. Some examples are: Australia, Belgium,

[16] `http://jarus-rpas.org/terms-reference`

Brazil, Canada, Colombia, France, Japan, Nigeria, Russia, Rwanda, South Africa, Switzerland, Ukraine, and the USA.

JARUS proposes the SORA approach as a guideline to National Aviation Authorities and to any possible applicant to a UAS certification. It consists of ten steps, involving the description of the Concept of Operations (ConOps), the determination of a Ground Risk Class (scaling from 1 to 10, related to the energy released in an impact on the ground), the determination of the Air Risk Class (related to which airspace is used), risk mitigation, and the determination of a Specific Assurance and Integrity Level (SAIL) (ranging from I to VI) and the respective Operational Safety Objectives (OSOs). The process is summarized in Figure 1.11.

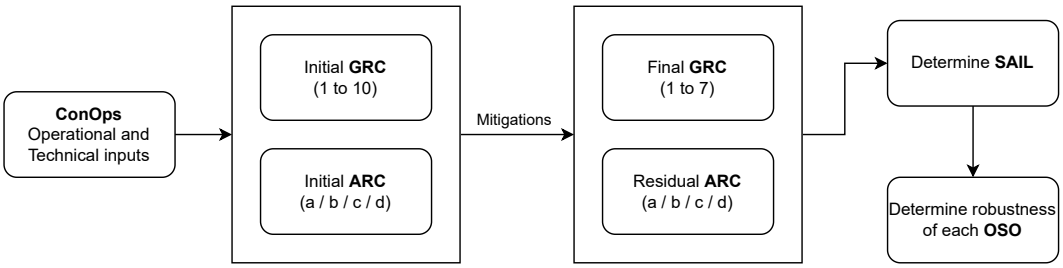

Figure 1.11: Summary of the SORA approach

It also proposes the bow-tie model for risk analysis, where a central event (the bow-tie node) is the undesired effect, on its left side its possible causes are shown (and their own causes as well), and on the right side the possible consequences are shown (and their own possible consequences). The bow-tie method eases risk identification and mitigation.

Nonetheless, whichever national standard we consider, the autopilot (flight stack) is treated as an electronic component, and no notion of software safety is present. Therefore, we need to look for similar fields and their software safety aspects, so we can propose a safe software architecture for drones. The automotive and the aviation fields were researched in order to provide the necessary inspirations.

1.4.2 Safety in Automotive Systems

The International Organization for Standardization (ISO) defined the standard 26262 entitled "Road vehicles – Functional safety" in 2011. It is "intended to be applied to safety-related systems that include one or more electrical and/or electronic systems and that are installed in series production road vehicles" [Iso]. ISO 26262 is an adaptation of IEC 61508, published by the International Electrotechnical Commission. Both these safety standards are risk-based, i.e., the risk of potentially dangerous situations is qualitatively assessed, and safety measures are taken with the objective of minimizing systematic failures or mitigating their effects.

ISO 26262 provides standards for the whole lifecycle of the vehicles, including design, production and operation. The standard involves a series of steps, starting by identifying the

item to be analysed. An item is a particular system or set of systems of the vehicle for which the standard will be applied. The item's functional requirements are defined, as well as a set of hazardous events that relate to the item. For each hazardous event, an Automotive Safety Integrity Level (ASIL) is assigned, and safety goals are derived accordingly.

The ASIL is a risk classification that will determine the adequate approach to deal with different potential hazards. It spans from ASIL A (the lightest constraints) to ASIL D (the most stringent constraints) – also, a QM level is defined, where no constraints are imposed since no unreasonable risk takes place. Determining the ASIL involves evaluating the risk according to three dimensions:

Severity: Relates to the potential injuries caused by the hazard. "No injuries" represents a severity level equivalent to S0; "moderate injuries" (survival is practically certain) represents level S1; "severe injuries" (probable survival) represents level S2; and "life-threatening (uncertain survival) or fatal injuries" represents a severity level S3.

Exposure: How likely the related injury can occur. If it is highly unlikely, it is classified as E0; a very low probability results in classification as E1; low probability, E2; medium, E3; and if the injury happens under most operating conditions, E4.

Controllability: Relates to the chance of the driver being able to act or react to the situation in order to prevent the injury. If the situation is generally controllable, the level C0 is applied; if it is simply controllable, C1; if most drivers could prevent the injury, C2; otherwise (difficult or impossible to control), C3.

ASIL D refers to the combination S3 + E4 + C3. Then, for every reduction in any classification, there is a one-level reduction in the ASIL classification (except when reducing from C1 to C0). For example, a combination S3 + E2 + C2 results from reducing twice the level of exposure and once the level of controllability, which means there are three reductions to the ASIL classification: it is therefore ASIL A.

Complementary to the risk-based approach of ISO 26262, the Automotive Open System Architecture (AUTOSAR) standard [Für+09] provides the means to separate the software analysis from the hardware. It proposes a common interface such that the same algorithm can run on any AUTOSAR-compliant electronic controller, regardless of the provider. In addition, it defines specific allowed variable names for each item in a vehicle system, *prescriptively* standardizing software development for the automotive industry.

1.4.3 Safety in Civil Avionics

At first, in order to guarantee safety aspects of the avionic systems in aeroplanes (such as redundancy), federated architectures were used. In other words, each function had to be executed in its own piece of hardware, so that a potential malfunction in one single function would not propagate to the others. However, this method does not scale when we consider more and more complex avionic systems, and an alternative approach to guarantee the isolation of malfunctions was found: Integrated Modular Avionics (IMA) [Pri92].

Instead of separating each function in a single processor, IMA creates statically isolated (in time and space) partitions in each piece of hardware. Furthermore, it allows the use of a network instead of direct connections between each physical component. IMA allows, in consequence, applications with different criticality levels to be executed in the same module. Therefore, it reduces the need for equipments, increasing the available weight and volume for other systems, cargo or passengers.

ARINC 653 defines the interface that allows IMAs to be implemented. It defines the Application Executive (APEX), which offers a standard interface for the Real-Time Operating System (RTOS) such that partitions can be statically defined. Standardizing the interface allows for a high degree of independence between the application algorithm and its implementation, as well as application portability between ARINC 653 compliant platforms.

Also, the standard DO-178C / ED-12C, entitled "Software Considerations in Airborne Systems and Equipment Certification" [AR92], is the required standard for software-based aerospace systems among certification authorities such as the Federal Aviation Administration (FAA) and the European Union Aviation Safety Agency (EASA). Similar to ISO 26262, it defines categories of criticality according to the possible consequences of a module failure. The categories are called Software Level, or DAL, and they range from level A (the most strict) to E (the most relaxed)

If a failure in the system may cause deaths and normally results in the loss of the aeroplane, the failure potential consequences are considered *catastrophic* and a DAL A is assigned, and a failure rate of at most 10^{-9} per hour is accepted. The failure is considered *hazardous* if it reduces the ability of the crew to operate the aircraft (as a consequence of substantial distress or higher workload), if it causes serious or fatal injuries to passengers, or if it has any other large negative effect on safety or performance; in this case, a DAL B is assigned and a failure rate of at most 10^{-7} per hour is accepted. A *major* failure significantly increases the crew workload, significantly reduces the safety margin, or may result in passenger discomfort or minor injuries – for which case a DAL C is assigned and the maximum accepted failure rate is 10^{-5} per hour. If a failure only slightly reduces the safety margins, slightly increases crew workload, or causes minor inconveniences to the passengers, it is classified as *minor* and is related to DAL D – at most 10^{-3} failures per hour are accepted. If no safety effect comes from the failure, DAL E is assigned. This is summarized in Table 1.3.

Failure type	**DAL**	**Maximum accepted failure rate**
Catastrophic	A	10^{-9}/h
Hazardous	B	10^{-7}/h
Major	C	10^{-5}/h
Minor	D	10^{-3}/h
No effect	E	-

Table 1.3: Relationship between DALs and failure types and accepted rates

According to each DAL, a set of specific safety objectives is defined (except for DAL E, where no objective is defined). Depending on the DAL, some of these objectives require an

independent team from the development one in order to guarantee their accomplishment. For DAL D software and above, the 26 objectives to be reached include: "High-level requirements are developed", "High-level requirements are traceable to system requirements", "Software architecture is developed" and "Executable Object code and Parameter Data Item Files, if any, are produced and loaded in the target computer." For software that was labelled as DAL C or above, in addition to the previously mentioned objectives, the 36 new objectives include, for example: "High-level requirements are verifiable", "Software architecture conforms to standards" and "Source code conforms to standards." For DAL B software and above, 7 new objectives are defined, including: "High-level requirements are compatible with target computer", "Source code is verifiable", and "Test coverage of software structure (decision coverage) is achieved." For DAL A, only two new objectives are added: "Test coverage of software structure (modified condition/decision) is achieved" and "Verification of additional code, that cannot be traced to Source code, is achieved."

Complementary to DO-178C, the Motor Industry Software Reliability Association (MISRA) standards for embedded C and C++ (MISRA C and MISRA C++) [Ass+04] were published as a set of guidelines recommending the use of "a restricted subset of a standardized structured language" for producing embedded software. They are intended to remove or at least reduce the opportunity to make mistakes when coding in C/C++. The focus is on the aspects of C/C++ that impact the safety or security of systems by improving reliability, readability, portability and maintainability [BBH18].

The standards seek to avoid compiler differences such as integer sizes, which can vary according to the processor used. They also seek to avoid the use of functions more likely to fail, such as "malloc". In addition, they promote maintainability with, for example, the use of naming conventions and comments. They also endorse testability by limiting the complexity of functions.

In contrast to AUTOSAR, it is *descriptive*, meaning it does not define strongly what must be done, but rather indicates a possible manner of reaching the objectives. There is still margin for changes and customization of the code.

1.5 COMP4DRONES

In this complex technological and regulatory context, a European company interested in using drones to inspect large areas (transmission lines, for example) has some possible solutions. The simplest one is to acquire a commercially available drone and use it as a COTS. The drone would be commanded by a pilot, and it would need to stay in their Visual Line of Sight (VLOS). Since the VLOS can be considerably short in comparison with the total surface area to be inspected, several flights would need to take place to cover the whole area, which makes the cost of the operation rise to the point where it is not competitive relative to a traditional approach (human inspection). Alternatives to this method would require sufficient autonomy and respective validation to allow for operations beyond VLOS, which demand time-consuming development efforts from specialized workers and a substantial investment to

certify the technology – once again, rarely competitive.

In order to provide more competitive and efficient options, the COMP4DRONES project took place. COMP4DRONES is a European project consisting of around 50 partners from both the public and the private sectors, "with the aim of providing a framework of key enabling technologies for safe and autonomous drones" [JU19]. Its goal is "to bear a holistically designed ecosystem from application to electronic components, realized as a tightly integrated multivendor and compositional UAV embedded architecture solution and a toolchain complementing the compositional architecture principles" [Nou+19].

Funded by the European funding programme Horizon 2020, COMP4DRONES presents its results in five different Use Casess (UCs):

UC1 - Transport: When incidents or scheduled events take place in unmonitored areas (lack of infrastructure or places hard to access), drones can quickly provide a real-time clear view of the current situation. This might allow an improvement of traffic management, a better monitoring of road conditions and faster response to accidents. Demonstrators include the domains of traffic management, port operations and railway infrastructure.

UC2 - Construction: By accessing difficult and hazardous areas, drones can provide valuable insight for the construction industry without the need to put human lives in danger. Two demonstrators explore this domain: the digitalization of a constructive process, and the analysis of underground constructions.

UC3 - Logistics: Drones can potentially reduce delivery costs, access remote places and increase delivery speed. Two demonstrators involve the Logistics domain: delivering geophysical sensors using autonomous UAV fleets, and delivering medical samples between medical facilities using both UAVs and Unmanned Ground Vehicles.

UC4 - Surveillance/Inspection: In harsh environments such as some industrial installations and disaster areas, using drones represents an advantage when compared to putting human lives at risk. Two demonstrators will represent this UC: the inspection of offshore turbines, and a fleet of robots exploring an unknown environment.

UC5 - Agriculture: Precision farming – a concept increasingly important due to climate change and the rapid growth of human population – can be made viable by the use of drones equipped with the proper sensors. Two demonstrators are envisaged: one regarding wide crop production and plant monitoring (focused on health and growth management), and the other regarding viticulture in remote areas with poor infrastructure.

In order to achieve the predicted goals, the project is divided into eight Work Packages (WPs):

WP1: Case studies, benchmarking and quality assurance;
WP2: Specifications and methodology;
WP3: Integrated modular architecture;
WP4: Drone autonomy;
WP5: Trusted communication;

WP6: Design, performance and verification tools
WP7: Exploitation, dissemination, training and standardization;
WP8: Project, risk and innovation management.

This work takes part in WPs 3, 6 and 7, as it involves an investigation of the modular architecture in autopilots, verification tooling and dissemination (in the form of scientific publications). The goals of this work, in the context of COMP4DRONES, is to reduce development costs by enabling reuse of safe modules. Instead of redeveloping functions for every different application, the developer should be able to use modules already developed and individually validated by other teams, as well as to easily assess the safeness of the system (composition of modules).

Hence, the approach sought in this work is to transform COTS into MOTS. This means that, instead of providing ready-to-use black-box components for drones, providers should develop ready-to-use "grey-box" components and modules. These would allow a "plug and play" approach, but also expose their inner functioning and properties to accept adaptations to individual needs. In addition, tools should be provided to analyse key properties of the resulting version of each module, as well as the system as a whole.

Part I

Processor and Bandwidth Overload Mitigation

Chapter 2

Background on Real-Time Programming

The high complexity of desired drone behaviours, in addition to weight and energy restrictions present in the domain of flying vehicles, demand the use of sufficiently powerful (high-speed calculations) and efficient (low-energy consumption) processors as the parts responsible for the drone's autonomy. Processors provide a sufficiently versatile base that can be reused for a wide range of applications, in contrast with the rigidity of any analogical system.

On the other hand, they operate in discrete time, while most theoretical control techniques and analogic systems are based on continuous time. However, by repeating discrete operations with a sufficiently high frequency, processors are able to satisfy the imposed stability constraints. With the intrinsic linearity and discreteness of processor operations, some strategies need to be taken into account to allow for the continuous repetition of distinct algorithms.

2.1 Definitions

2.1.1 Basic concepts

The vast literature in real-time programming, robotics, aerospace and control theory contain a considerable amount of terms that not always describe the same elements. In order to mitigate any possible ambiguity, the following definitions are used in this thesis, supported by representations in Architecture Analysis and Design Language (AADL) (better explained in subsection 5.2.1).

System

Definition 2.1. A system is any composition of sub-systems and elements that function as a whole.

The AADL representation of a system is presented in Figure 2.1.

Examples of systems are a drone, a drone autopilot, a UAS, etc. The focus of this thesis are physical systems that contain a processor as its critical component (or as one of them).

Figure 2.1: AADL representation of a System

Processor

Definition 2.2. A processor is a logically programmable entity. It is capable of executing an algorithm, one elementary instruction at a time, including conditional and non-conditional jumps.

A processor is represented in AADL as in Figure 2.2.

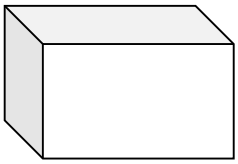

Figure 2.2: AADL representation of a Processor

Some examples of processors are microcontrollers, microprocessors, a Digital Signal Processor (DSP), etc. They are responsible for controlling the system such that it performs a certain desired behaviour.

Behaviour

Definition 2.3. A behaviour is the set of actions and decisions a system makes.

Some examples of system behaviours, also sometimes called "skills", are (in the robotics domain): serving coffee, delivering packages, detection and avoidance, etc. These behaviours can be reached by the development of algorithms that implement them.

Algorithm

Definition 2.4. An algorithm is a sequence of instructions.

While the algorithm is a more abstract entity (such as a cake recipe, for example), it can be implemented by a concrete entity: a program.

Program

Definition 2.5. A program is an executable entity of computer logic, implementing one or more algorithms.

Programs are storable in the computer memory, ready to be used by the computer's processor.

Function

Definition 2.6. A function is a subdivision of a program, i.e., a sub-program. Each function implements a specific algorithm, which is a modular piece of the whole program. Functions might have a set of inputs, a set of outputs and a set of configurable parameters.

Executing a function requires the existence of a function call stack (or simply "stack"). The stack is a linear data structure that keeps record of local variables and the active functions of a program.

2.1.2 Embedded systems

Historically, embedded software was conceived using the strategy of *cyclic executives* [Lui+04]. These are infinite loops that sequentially call functions to be executed in a statically defined order. Cyclic executives allow the use of basically three different strategies regarding the time at which each function is called:

1. Calling functions as soon as possible, with no notion of time in the algorithm. Consequently, the processor is always at 100% of utilization, and no periodicity is present (since each function executes in a variable amount of time).
2. Using the hardware specific drivers to determine a fixed amount of time to wait between function calls. This reduces the use of the processor, but due to variations in the execution time of each function, there is often no way to guarantee any strict periodicity in the system.
3. Using the hardware specific drivers to determine instants in time after which the loop can continue its execution (instead of a fixed waiting time). In other words, the system's clock is read at the beginning of the main loop, and in case its value is bigger than the last execution instant of a function plus its period, the function is executed and its last execution instant is updated. This allows for a best-effort periodicity, where the execution of the system will be more easily predictable.

In a complementary manner, the frequency of certain functions can be reduced by using counters, such that each execution of the `main` loop increments the counter, and the respective function only gets called when the counter reaches its desired value. Then, the counter is reset to its initial value. Figure 2.3 shows an example of such a strategy, where each function has a period (the inverse of the frequency) equal to an integer multiple of the main period.

With the development of architectures where it is possible to interrupt (preempt) and schedule programs according to pre-defined priority rules, new concepts emerged, notably the concept of a thread.

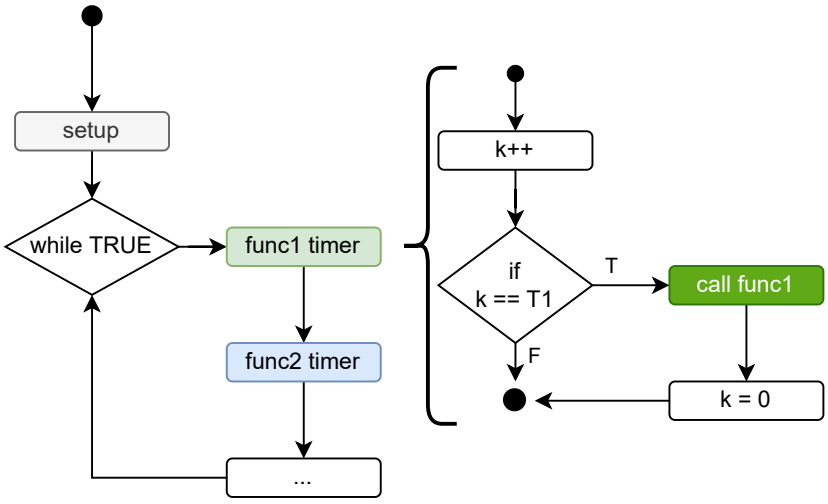

Figure 2.3: Flowchart of an example of a cyclic executive

Thread

Definition 2.7. A thread (represented by the Greek letter τ, often with an index) is an execution chain, a sequence of instructions in a specific processor and memory context (analogous to a train of thought). Each thread has its own stack (a register of its state).

Every program needs to be executed in a thread. Threads might also be referred to as "tasks", a term that we avoid due to the possible meaning more closely related to a goal yet to be accomplished, more similar to our definition of "behaviour". In the AADL language, a thread is represented as depicted in Figure 2.4.

Figure 2.4: AADL representation of a Thread

Usually, threads are recurring, meaning that they loop indefinitely, going through the same main function each time they loop (although this main function can call different sets of functions depending on the input data).

Job

Definition 2.8. A thread job is one loop, one unit of repetition of a recurring thread.

Thread jobs are *released* by an external (e.g., input signals) or internal (e.g., system clock) trigger, and the thread will be executed at that moment if there are no other threads waiting to be executed. Threads can be designed to be *periodic*, meaning they will try to execute each

job in a regular time interval, defined by its frequency (notably using the internal clock). An alternative design is that of a *sporadic* thread, where jobs do not have a determined instant in time when they are released (notably using external signals) – although they have a minimum separation between two jobs, therefore having a maximum frequency.

Period

Definition 2.9. The period T_i of a thread τ_i is the distance in time between any two successive job releases in the case of periodic threads (i.e., the inverse of the thread's frequency), or the minimum distance in time between two successive job releases in the case of sporadic threads (the inverse of the thread's maximum frequency).

When a system where several periodic events take place, this system will also present a period, an interval of time that repeats itself until infinity.

Hyperperiod

Definition 2.10. The hyperperiod (H) is the amount of time it takes for a schedule to return to the same state of reference. It is mathematically equivalent to the Least Common Multiple (LCM) of all thread periods.

With the need to prioritize certain instructions with respect to less important algorithms, processors able to instantaneously switch between different threads were developed. They are able to register the current progression in the execution of a program (calling stack), as well as the data currently loaded in the processor, so that any thread can be "paused" and resumed later. These interruptions allow the implementation of multithread architectures.

Multithread

Definition 2.11. A multithread system consists of a system where several threads can be executed in parallel, even though a single processor might be in use.

In a multithread architecture, a thread, once initialized, can therefore be in at least three states, as represented in Figure 2.5:

Executing: when its instructions are being executed by the processor;
Idle: when its current job is done, and the thread is waiting for the next job to be released, leaving room in the processor for other threads to be executed;
Ready: when its current job is not yet done, but the processor is occupied by other threads. The job will resume once higher priority jobs are done.

A multithread processor might be implemented in a *collaborative* manner (each thread needs to actively give space to the next executing thread) or a *preemptive* manner (threads can be interrupted by an external entity at any given instant). As an example of an external entity, we have the scheduler. However, to introduce the scheduler, we have to introduce first the Operating System (OS), since the implementation of a scheduler needs to be made with the use of an OS.

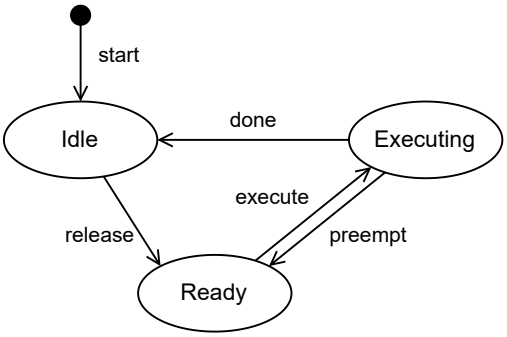

Figure 2.5: Simplified state machine representing thread states

Operating System

Definition 2.12. An Operating System (OS) is an abstraction layer that provides a unique common programming interface to several different hardware pieces. It also provides a set of basic functions and a scheduler.

An example of an OS is Linux, which is open-source and used in projects such as the first Mars flying drone, Ingenuity[1].

Without an OS, a system is said to be *bare-metal*. The bare-metal approach presents some limitations: the function code is often dependent on the specific platform it uses (specially for input/output devices); writing new functions to the system often requires the knowledge of how all the functions are organized, going against principles of modularity; also, other scheduling techniques are better at guaranteeing the completion of each function in its defined deadline. Therefore, using an OS can provide some advantages, although it has its drawbacks: determinism is reduced, therefore efforts to certify the system need to be larger.

A General Purpose Operating System (GPOS), intended to be used for almost any kind of application, generally contains non-deterministic schedulers. Their schedulers operate according to fairness policies, which are better at providing a fast response to multiple interactive applications, but do not distinguish between high-priority threads and low-priority ones. Fairness policies are, therefore, not suitable for critical applications.

RTOSs, on the other hand, have been designed according to the needs of real-time systems. Their schedulers are able to prioritize threads according to a statically defined convention or an instantaneous condition (e.g., which thread was released first). While this approach is less suitable for interactive applications (mostly used in GPOSs), it is a better approach for avoiding deadline misses in critical domains. Some examples of RTOSs are FreeRTOS[2], NuttX[3] (used in PX4) and ChibiOS[4] (used in more recent version of ArduPilot and Paparazzi).

[1] `https://spectrum.ieee.org/nasa-designed-perseverance-helicopter-rover-fly-autonomously-mars`
[2] `https://www.freertos.org/`
[3] `https://nuttx.apache.org/`
[4] `https://www.chibios.org/dokuwiki/doku.php`

OSs have, as a constituent part, a Hardware Abstraction Layer (HAL). It consists of a common interface for the higher-level applications and any supported hardware. Using distinct compilation targets according to the target hardware, it uses the proper hardware-specific functions and drivers to compile the same application code into distinct processors. Therefore, the application development efforts can become independent of hardware specificities.

A representation of the abstraction provided by the HAL is given in Figure 2.6. Each possible implementation (combination of hardware and embedded software) can be seen as the assembly of the elements crossed by a certain vertical line in the image. For example, in order to embed the application code in the hardware B, its specific drives are used, and a specific adaptation of the HAL for the hardware B is present, but the HAL and the application code are the same as if hardware A or C was used.

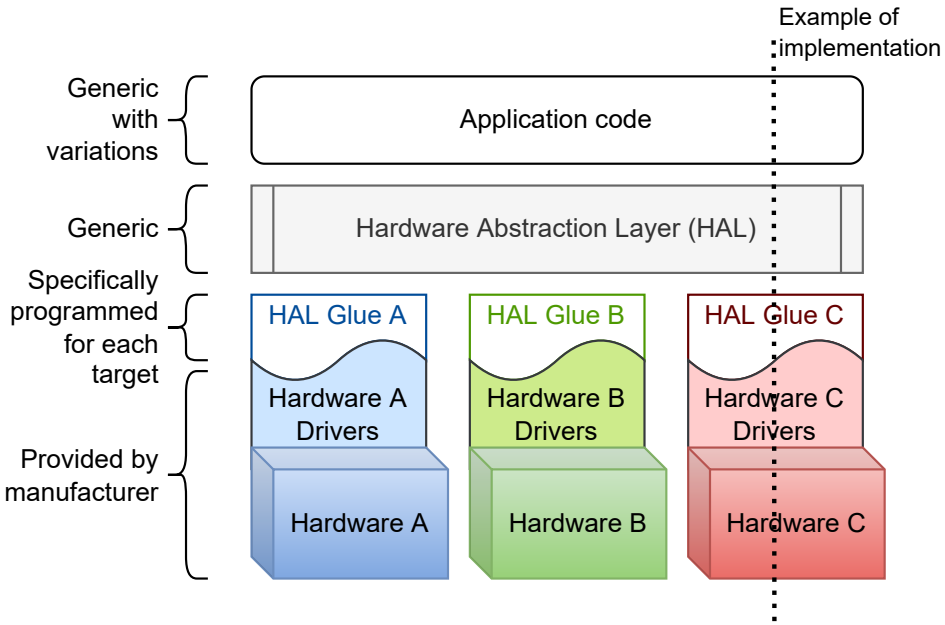

Figure 2.6: Representation of the HAL abstraction

When an OS is multithread, although each thread has its own stack, they can be given access to the same memory space. This means that it is possible for different threads to read and write values for the same variables (herein called *global variables*), and the actions performed by one thread can affect the others. Therefore, thread interruptions can eventually generate unexpected behaviours. For example, considering a global variable x with any integer initial value, if a thread τ_A is responsible for reading its value and then replacing it by its successor $(x+1)$, while thread τ_B reads x and, if its value is even, it divides it by 2 (otherwise it does nothing), we expect x to always be an integer. However, the *race condition* between τ_A and τ_B can result in an interruption of τ_B by τ_A right after τ_B identifies x as even, causing x to be incremented by 1 just before it is divided by 2 by τ_B. In that case, x would not be an integer anymore.

In order to isolate certain groups of threads such that threads in different groups can never share variables with another group, some OSs allow the creation of processes, guaranteeing a higher level of modularity.

Process

Definition 2.13. A process is an abstraction of a memory isolation technique that blocks, by construction, different sets of threads from having access to the same memory space.

Processes are represented, in AADL, as in Figure 2.7. They contain one or several threads.

Figure 2.7: AADL representation of a process

Although threads in the same process can eventually all use the same memory space to either read or write data, they cannot access the same memory as threads in a different process – that memory will be isolated from the first process as if the second process happened in a different computer. However, if two threads need to share information such that no race condition occurs, resources such as mutexes and semaphores need to be used in order to keep the system thread-safe.

Scheduler

Definition 2.14. The scheduler defines the order of execution of each thread, interrupting those in execution in case needed.

Schedulers can operate according to different scheduling policies, such as Preemptive Fixed Priority (P-FP), Non-preemptive Fixed Priority (NP-FP), First-In-First-Out (FIFO), etc. For P-FP schedulers, only higher-priority threads can pre-empt a thread already in execution. In the case of NP-FP, threads are put in a queue such that the next thread to be executed is the one with the highest priority. Higher priorities can be given to threads with shorter periods (Rate Monotonic), to threads with more strict constraints (Earliest Deadline First), or according to any other arbitration. FIFO schedulers, however, execute threads in the order they are released – a simple non-preemptive scheduler to implement, widely used in many applications. Schedulers are further detailed in section 2.2, as some aspects of real-time programming need to be developed first.

2.1.3 In a Real-Time Context

With the objects defined previously, several policies and analyses can be made to extract important properties of real-time systems. They provide the necessary tools to prove (or

disprove) the safety of a system in early stages of its development. The most important concepts for this thesis are explored in the following paragraphs.

Knowing the scheduling policy of a given system, it is possible to calculate some of its properties, such as the maximum *response time* of a given thread.

Response Time

Definition 2.15. The response time of a job in a schedule is the time interval between a job release and the instant in which the job finishes its execution, allowing the thread to return to an idle state or to a ready state for a following job.

The maximum response time, being it an upper boundary to the time it takes for the effect of a certain thread to take place, is fundamental to the safety analysis of any system. Given that threads often have timing constraints, either due to hardware limitations or to parameters related to the system's stability, the comparison between computed values for the maximum response times and these constraints allows the system designer to conclude if the system is feasible before its construction.

Deadline

Definition 2.16. A deadline is the maximum tolerable response time of a thread.

Usually, we consider that systems have *implicit deadlines*, i.e., the deadline of a thread is equivalent to the thread's period.

For calculating the maximum response time of a thread, its execution time has to be taken into account. Since it is rare that threads have a fixed execution time, given that the number of operations often varies according to the inputs, we need to consider the worst-case scenario, which is frequently when the execution time is the largest possible.

WCET

Definition 2.17. The Worst Case Execution Time (WCET) of a thread τ_i, represented by C_i, is the maximum expected value for the amount of time a job needs to use the processor to finish its execution.

An important metric in analysing real-time systems is the processor utilization, which can be calculated with little effort but can already indicate cases where it is impossible to schedule the system.

Processor Utilization

Definition 2.18. The processor utilization, represented by U, is given by Equation 2.1, for all threads τ_i in the system.

$$U = \sum_i \frac{C_i}{T_i} \tag{2.1}$$

For example, if a system containing a single processor has a utilization factor $U > 1$, we know that there cannot be any algorithm capable of scheduling it such that it respects its deadlines, since the processor will always accumulate work to be done. However, for systems with $U \leq 1$, some might be scheduled while others might still be deemed to missing their deadlines.

In order to analyse if a system meets its deadlines or not under a certain scheduling algorithm, we can either try to simulate every possible situation in the schedule, or we can look for a worst-case configuration in the execution pattern. This worst-case configuration is the critical instant.

Critical instant

Definition 2.19. The critical instant of a schedule is the possible execution pattern that will generate the maximum response time of a specific thread.

Often, the critical instant is the moment when every thread is released at the same time (*synchronously*). However, the critical instant may be different for different threads in the same system, specially if threads are configured so they are never released synchronously.

In order to avoid critical instants, several strategies have been developed, either by the use of different schedulers or by the configuration of certain thread systems, as it is described in the next sections.

2.2 Scheduling Strategies

At first, fixed priority schedulers were created [Lui+04]. In these systems, each thread is statically given a priority that will determine, in case two threads are ready at the same time, which one will be executed first. An example of fixed priority scheduling is the Rate-Monotonic, which assigns the highest priority to the thread with the shortest period. Also, the Earliest Deadline First (EDF) protocol defines the highest priority to the thread with the smallest deadline.

Furthermore, dynamic priority schedulers were developed, such as FIFO. As the name indicates, the first thread to be released in a FIFO scheduler is the first to be executed – and, eventually, there can be a tie-breaking priority assignment such that the scheduler can decide the order of execution in the case of a simultaneous thread release.

In addition to the priority, the aspect of *preemptiveness* was introduced. If a scheduler can "pause" the execution of a thread, it is said to be preemptive. Fixed-priority schedulers have their preemptive and their non-preemptive variations, while the FIFO scheduler, by definition, can only be non-preemptive.

According to [ASN16], FIFO schedulers present several advantages in embedded systems, notably for having a simple implementation and for allowing the predictability of the schedule. Firstly, FIFO schedulers are work-conserving, meaning they do not add any busy time to the processor (in contrast to, for example, possibly waiting for threads to be released, a strategy used by some schedulers such as in [NF16]). Also, FIFO schedulers are c-sustainable, i.e, any schedulable thread set will remain schedulable for any arbitrary reduction in execution times. This property is crucial, considering that threads often have a WCET, but the exact execution time is unknown. In addition, in the case of periodic threads (not sporadic) and a deterministic tie-breaker, FIFO schedulers are completely deterministic.

Additionally, not only some thread systems are scheduled under a FIFO protocol, but also several communication drivers that schedule messages in a given communication link. It is common to implement communication links as a FIFO queue that receives data from internal components and send them individually over the communication bus, following the "first come, first served" logic.

Furthermore, in order to avoid an instant where every thread is released at the same time, it is common to shift the initial instant when a thread is released. This strategy is well explored in the scientific literature, as shown in the next section.

2.3 Offset Free Systems

An offset free system, as first defined in [GD97], can be thought of as a set of periodic events with finite duration that start happening at an arbitrary instant in time: the offset. In addition, offsets are configurable, allowing the search for an optimal configuration.

Offsets

Definition 2.20. Offsets are an amount of time a periodic thread is configured to wait before it begins its pattern of job releases.

Without offsets, jobs from any thread τ_i are released at the instants $r_{i,j} = j \cdot T_i$ $(\forall j \in \mathbb{N})$, while an offset O_i will make every job be released in the instants $r_{i,j} = O_i + j \cdot T_i$ $(\forall j \in \mathbb{N})$. The right choice of offsets might prevent a critical instant from happening without changing the system architecture (only its configuration), demonstrating the interest in studying this kind of solution.

Important properties can be extracted from offset free systems. For example, the authors in [PL05] derive the minimum separation between the release times of two periodic threads, as stated (in a different mathematical notation) in Theorem 2.3.1.

Theorem 2.3.1
Given two concrete threads τ_i and τ_j (where a concrete thread is simply a thread for which there is a specific starting time – the offset – as first introduced in [JSM91]), with respectively O_i and O_j as their offsets and T_i and T_j as their periods, the minimum distance between the release of a job of τ_i and the next job of τ_j is given by:

$$\Delta_{i,j} = (O_j - O_i) \bmod \mathrm{GCD}(T_i, T_j) \tag{2.2}$$

In Figure 2.8, a visual representation of Theorem 2.3.1 is presented, inspired by the visualization proposed in [Gro99]. From the figure, it can be seen that, if $\Delta_{i,j} < C_i$, in the worst case scenario, the execution of τ_i delays the execution of τ_j. Analogously, τ_j delays the execution of τ_i in the worst case scenario, if $\Delta_{j,i} < C_j$.

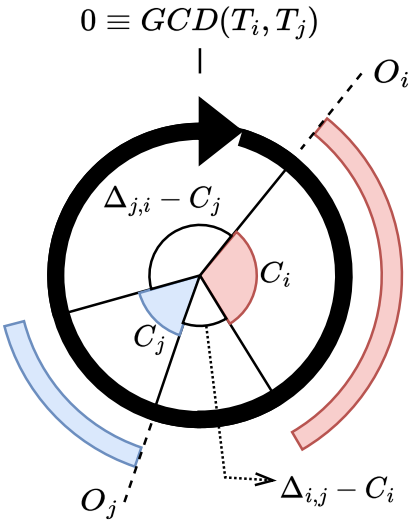

Figure 2.8: Visualization of Equation 2.2

The figure resembles the visualization of thread systems that can be seen in [Yom09]. However, instead of representing the modular circle with a size equal to the LCM of the thread periods, here the modular circle has a size equal to the Greatest Common Divisor (GCD) of the thread periods. This choice is made in order to represent not every job release in the hyperperiod, but only the cases where these job releases are the closest (the most critical case).

It is also important to notice that, as a consequence of Equation 2.2, for every set of periodic threads, if every possible pair of different threads τ_i and τ_j respect the conditions that $\Delta_{i,j} \geq C_i$ and $\Delta_{j,i} \geq C_j$, then there is no delay in the execution of any job: each thread will have, as its maximum response time, simply its maximum execution time.

In the same paper, the authors prove that the number N of distinct offset assignments is,

for each offset free system of n threads, given by Equation 2.3.

$$N = \frac{\prod_{i=1}^{n} T_i}{H} \tag{2.3}$$

Equation 2.3 shows that, although divided by the hyperperiod H, the number of possible assignments tends to a combinatorial explosion with respect to the number of threads. Therefore, when choosing the offsets, most solutions rely either on heuristic methods and random or arbitrary values, or on optimization techniques that are considerably time-consuming.

For instance, it is possible to just randomly apply offsets to each element [ASN16], but other propositions take into account properties of the system in order to reach more optimal solutions.

The autopilot Paparazzi [BH08]; [Hat+22]; [Pap18]; [HBG14]; [Pap] assigns offsets to telemetry messages in a FIFO queue as such: the first message receives a null offset (zero); the second one receives an offset equivalent to 10% of its period; the third one, 20%; and so on, until 90%; and the next message receives again a null offset; and the cycle repeats until every message has its own offset.

In [Goo03], an algorithm (hereafter called "Goossens Heuristics") was proposed to order every possible thread pair in a set according to a decreasing order of the GCD of their periods. The objective is that each pair of threads (starting from the highest GCD) is given a pair of offsets which are apart by half of their GCD. In other words, for each unassigned pair of threads τ_i and τ_j, τ_i gets a random offset O_i while τ_j gets $O_i + \frac{1}{2}\text{GCD}(T_i, T_j)$. Therefore, the algorithm tries to separate thread releases as much as possible. The complexity of the algorithm is $\mathcal{O}(n^2 \cdot (\log T_{max} + \log n^2))$.

We can also think of a slight modification to this method (hereafter called "Modified Goossens Heuristics"): instead of creating a maximum separation of job releases, we can create a maximum separation of their "half instants" (release instant plus half of the execution time). This can be useful in cases such as, for example, threads whose periods have a GCD of 4 units, and one thread has an execution time of 3 units while the other has an execution time of 1 unit. The Goossens Heuristics would separate their release times by 2 units, creating an overlap of 1 unit, while accounting for their execution times would allow for reaching the optimal solution: separating release times by 1 unit.

In [GGN06], the previous method is adapted so the order of thread pairs to be evaluated is changed. In addition to the decreasing value of GCDs, four new orders are proposed, including the increasing value of GCDs. The principle of separating thread releases as much as possible is kept, and the algorithm complexity is the same.

Later, a method was proposed in [GHN08] (hereafter called "CAN Message Heuristics"). It looks for the longest least-charged time interval in the interval $[0, T_{max})$, assigning thread releases to the middle of that interval (starting from the thread with the lowest period). Its complexity is said to be $\mathcal{O}(n \cdot T_{max})$, therefore it is pseudo-polynomial with respect to the

period values T.

Specifically for FIFO schedulers, the authors in [NDB18] propose using one or several offsets per thread in order to reproduce the schedule constructed by a different scheduler (EDF, for example). This means that each thread job would be treated individually. Nevertheless, implementing this approach in the case of the Paparazzi autopilot, for example, would require the autopilot's code to be modified, retested and revalidated, while a single offset per thread requires only a specific configuration (already expected by the code).

These methods can be applied to finding offsets for independent thread systems or messages in a communication link due to the similarities in their mathematical models. However, real thread systems often present dependency constraints between threads due to sharing some critical resource. For example, a thread can be activated only after another thread outputs the results of its calculations. In order to schedule these systems, the "independent thread model" has to be updated, and therefore the most used strategies for communication between threads have to be analysed.

2.4 Communication strategies

Synchronous communication is a straight-forward solution. As an example of implementation, one function calls a second function once its output is calculated, as seen in Figure 2.9. The second function is executed as soon as the data is sent.

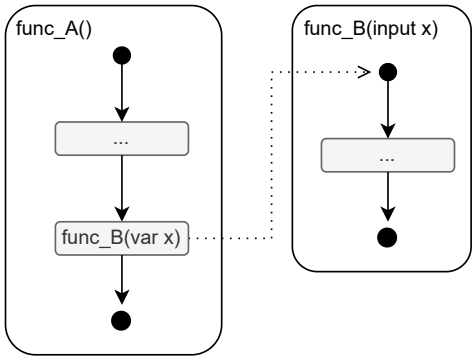

Figure 2.9: Diagram of an implementation of a synchronous communication pattern

In order to better schedule functions and allow for their modularity, an asynchronous approach is preferable. Each function will have its own rhythm, providing and assessing variables as it needs. However, if multiple modules try to access the same variable at the same time, undesired effects might arise and corrupt the data (race condition). Some strategies need to be used in order to avoid these effects.

If a common variable is used, RTOSs can provide mutexes and semaphores. A mutex (short for "mutual exclusion") is a lock that allows only one thread to access a critical section at a time. A semaphore is a mechanism based on tokens, that are produced and consumed

by the threads that access the critical instants. Using these resources, several strategies can be implemented, such as the publish-subscribe mechanism, where each thread can publish and/or subscribe to several variables at once.

While communicating thread patterns can be implemented only using basic software concepts, if two threads are in separate processes, the OS needs to be considered for sharing information. Inter-Process Communications (IPCs) allow sharing data between two different processes in the same computer, or even different connected computers.

Specifically for distinct hardware pieces, communication buses allow the transport of messages over a certain distance, either by using cables or wireless communications. Either way, they need to use peripherals responsible for serializing the data into a protocol-compliant form of the message. Mostly used protocols in the drone domain are CAN, UDP, UART and Ethernet. Each protocol has its own patterns for defining headers, tails and the serialization of the data.

To better understand the communication between threads in different computers, we can rely on the Open Systems Interconnection (OSI) model [DZ83]. This model proposes an abstraction of communication strategies based on the identification of seven layers (in increasing degree of abstraction):

1. **Physical:** the physical connection between communicating entities (a copper cable, electromagnetic frequencies, etc.);
2. **Data Link:** responsible for managing the transmission of data between two directly connected communicating entities;
3. **Network:** responsible for managing networks, i.e., several communicating entities interconnected, such as with the use of addresses;
4. **Transport:** responsible for the transmission of data segments over two entities in a network;
5. **Session:** responsible for the opening and the closing of communication sessions between two entities;
6. **Presentation:** responsible for data encoding and translation from the network to the applications;
7. **Application:** the high-level use of data, where threads and processes are.

Depending on whether the communicating modules are deployed in the same process or even in distinct computers, communication solely in the most abstract layer (Application) might be sufficient, or it might require low-level implementations as low as the Physical layer. This is why implementing a modular application can be challenging: the choice for the deployment of each module will potentially require an adaptation in the communication strategies used. Some deployment choices are depicted in Figure 2.10, complying to AADL notation. However, it is possible to leave the responsibility of implementing the suitable communication pattern to a module of its own: a middleware. Middlewares are applications that run on top of the OS and take charge of the communication between the modules built on top of it, so the application code can be the same even for different deployments. Some

notable examples of middlewares are ROS[5], ROS2[5] (which uses DDS/RTPS), uORB[6] (used in PX4), and ABI[7] (used in Paparazzi).

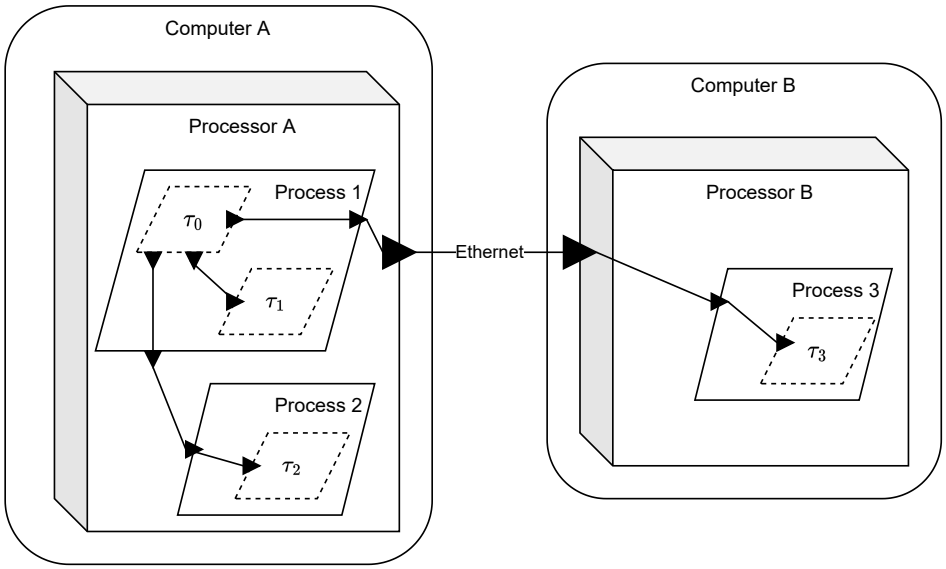

Figure 2.10: Example of thread deployments

ROS (acronym for Robot Operating System) is a middleware conceived initially for robotic applications. In such a system, functions are implemented in parallel nodes, and a master node keeps a list of currently active nodes. The master node is responsible for managing the communication between nodes, which is made using topics. A node can publish or subscribe to any desired topic, and the master node takes charge of calling the necessary functions to update each subscribing node. Although simple and efficient, ROS can only operate in multiple machines in some specific scenarios, all of which rely on a single master node (it becomes then a single critical point). In order to remove these constraints, ROS2 was developed.

Since ROS2 is based on very different communication principles of ROS, it was developed as a new middleware instead of an update to ROS. ROS2 uses DDS (acronym for Data Distribution Service) [PC03]; [Gro15], a standard developed by the Object Management Group (OMG) for distributed real-time systems. DDS is based on quality of service instead of strict conditions, and its wire protocol RTPS (Real-Time Publish-Subscribe) [Gro19a] allows it to be used in several machines in a network.

For some specific applications, *ad hoc* middlewares can be developed, such as ABI. Due to its application limited to Paparazzi autopilots, it was not made to support multithread communication, so it is said not to be thread safe. In contrast, uORB was developed such that different threads could safely use it to communicate, as it is used by threads in PX4.

To summarize, Table 2.1 presents the communication strategies and their possible appli-

[5] `https://docs.ros.org/`
[6] `https://docs.px4.io/v1.12/en/middleware/uorb.html`
[7] `https://wiki.paparazziuav.org/wiki/ABI`

cability. The columns are based on the structure presented in Figure 2.10, where the thread 0 communicates with every other thread.

Table 2.1: Comparison between applicability of different communication strategies

Strategy	**Thread 1**	**Thread 2**	**Thread 3**
Shared variable	✓	-	-
IPC	-	✓	✓
Communication bus	-	-	✓
Middlewares	✓	✓	✓

When threads need to share resources, their scheduling is more complicated. New constraints arise from the dependency on shared components, and more sophisticated techniques need to be used to analyse the execution of the system. One example is Semaphore Precedence Constraints (SPC) [Ngu+18]. This technique relies on the use of a single integer (positive or negative) to indicate a precedence relation between threads with equal or different periods, based on the mechanism of a semaphore.

In the case of equal periods, the concept of thread precedence is quite intuitive: a preceded thread job cannot execute until the preceding thread job is executed. Therefore, if τ_i precedes τ_j, the first job to be executed must be of τ_i, then τ_j executes, and the cycle repeats. On the other hand, if τ_i precedes τ_j such that, for example, $T_i = 3$ and $T_j = 5$, the notion of precedence is not intuitively clear. SPC treats this with an integer X such that the system behaves as if τ_i produced T_j tokens at each execution, τ_j would consume T_i tokens at each execution, X tokens would be available since the beginning, and τ_j can only execute if there is at least T_i available tokens. This simple set of rules allows the specification of any desired execution pattern between the two threads (as long as the pattern is compatible with the thread periods).

Once the communication between any pair of entities is made safe, the analysis of critical networks can be made. Although critical applications need to comply to real-time constraints, several applications nowadays use the Ethernet mechanism due to its simplicity and easily accessible components – but which is designed for maximizing throughput and minimizing average delays in a best-effort communication. This approach is not directly compatible with real-time constraints, and for that Ethernet needs to undergo an adaptation. TTEthernet and Time Sensitive Networks (TSNs) [COA17] have been able to bring real-time aspects to Ethernet communication by means of statically scheduling the network frames.

Involving several pairs of communicating agents simultaneously, critical network schedules quickly become too complex for any analysis that does not involve the use of powerful computational tools. Common approaches nowadays either use heuristic methods to propose a potentially satisfiable solution, or computationally expensive methods that seek an exact or even an optimal solution to the problems. These methods are detailed in the next section.

2.5 Constraint Programming and Optimization Techniques

If a problem can be expressed mathematically, but has no analytical solutions, and the solutions exist in a domain that is too large to be fully explored by hand, constraint programming and optimization tools might be very useful. These technologies rely on the large number of calculations that computers are able to perform per second to explore vast solution spaces, as well as on several techniques to safely reduce the size of the domains to be explored.

In Satisfiability Modulo Theories (SMT), a logical expression involving several boolean variables is considered. The expression is formed after a compilation of multiple logical constraints. The values for each variable are evaluated in order to look for any value set that would make the logical expression be true. Variables and constraints can also be in the form of mathematical equalities and inequalities, such that sets of numbers are searched to satisfy the final expression.

Considering the complexity and abundance of logical constraints in network scheduling, SMT has been used as an important technique to schedule traffic in critical networks [Ste10]. Among various available solvers, Z3 [MB08] (from Microsoft), CPLEX[8] (from IBM) and OR-Tools [Per11] (from Google) stand out.

Optimization Modulo Theories (OMT) is an extension to SMT, providing the possibility to look for solutions to the given set of constraints that optimize a certain objective. Naturally, OMT demands significantly more time to finish its calculations, since not only it needs to satisfy the set of constraints, but it must be a better solution than all the other possibilities. Z3, CPLEX and OR-Tools also allow the optimization of solutions.

[8] `https://www.ibm.com/analytics/cplex-optimizer`

Chapter 3

GCD+: Scheduling heuristics

By analysing the structure of the autopilots and the available scheduling strategies in the state of the art, we realized that it was necessary to deepen the understanding of the autopilot scheduling. The currently available scheduling strategies require either the autopilot scheduler to be rewritten, or that random or arbitrary calculations be made in order to define offsets and thread periods. These strategies have therefore considerable room for improvement.

We chose to base our analysis on the structure of the Paparazzi autopilot as a representative of all the frameworks. Furthermore, given the similarities between the functions in the autopilot cyclic scheduler and messages in a single communication link, the latter was chosen as the first case study – a simpler variation of the former, as messages are independent entities, while threads can communicate with each other.

Therefore, in this chapter, we analyse the problem of scheduling messages in the telemetry downlink of Paparazzi, represented by Figure 3.1[1]. The model applies for messages or threads, but for the sake of simplicity, we will only use the terms relative to thread systems.

Figure 3.1: Paparazzi UAV simplified telemetry overview

[1] https://wiki.paparazziuav.org/wiki/Overview

3.1 Motivating example

A thread system Φ is a set of n periodic threads executed on a single processor under FIFO scheduling (and, therefore, non-preemptive). The FIFO scheduling is often used for messages in a network communication link due to its simplicity, and the cyclic executive in an autopilot can be modelled roughly as a FIFO scheduler. Each thread τ_i' (such that $i \in [1, n] \subset \mathbb{N}$) is given by a tuple composed of two positive integers – its period T_i and its WCET C_i. Therefore, Φ is given by Equation 3.1.

$$\Phi \triangleq \{\tau_i' \mid \tau_i' = (T_i, C_i)\ \forall i \in [1, n]\} \tag{3.1}$$

Threads in the cyclic executive can be easily configured such that their initialization is offset relative to the autopilot startup instant. This means that they can be modelled as an offset free system, and Theorem 2.3.1 applies. Similarly, messages in a communication link can have a repeating pattern which is offset of the pattern starting on $t = 0$. In Paparazzi, this is done by setting the parameter "phase" in XML files that configure telemetry message parameters[2].

From Theorem 2.3.1, it can be seen that, in order to avoid delays, thread periods or offsets can be chosen in order to increase the minimum distance between job releases and, therefore, avoid critical instants. Considering that the period of a thread is influenced by several variables that go beyond the scope of our analysis (such as control stability), we will focus on choosing offsets to fulfil our goal.

We are interested in finding a vector of n integers O_i such that the threads in the concrete thread system Θ, defined by Equation 3.2, have no interference from other threads – i.e., every thread is executed as soon as it is released, no thread has to wait in the queue, and the system is schedulable under the strictly periodic thread model, where response times must equal the execution times of each job [MGS12]; [MS11]; [KS08]; [YS07]; [YS06]. However, our thread model is not strictly periodic: we do tolerate delays, yet we seek a configuration that gets as close as possible to the "schedulable under strict periodic constraints" case. This means that we are not primarily focused on a schedulability condition, but on reducing as much as possible the interferences between threads. In terms of messages, we seek the offsets that make no more than one message wait in the queue for transmission, or as close as possible to a schedule with no waiting times.

In order to find it and to guarantee data freshness, we seek a set of offsets that will make threads have the earliest starting time without overlapping into the execution of other threads. Since the scheduler is non-preemptive, an earlier starting time also means a lower response time.

$$\Theta \triangleq \{\tau_i \mid \tau_i = (\tau_i', O_i)\ \forall i \in [1, n]\} \tag{3.2}$$

According to Equation 2.3, there is an exponential number of possibilities to be considered.

[2] `https://github.com/paparazzi/paparazzi/blob/master/conf/telemetry/default_rotorcraft.xml`

Therefore, an analysis of the NP-completeness of the problem itself might shed light into the best option for finding a solution: if we should rather seek a general polynomial algorithm for solving the problem, or if no such algorithm exists (unless $P = NP$).

Amongst known NP-complete problems [GJ79], the "Simultaneous Incongruences" (SI) shares important aspects with the one evaluated in this chapter – a transformation from 3SAT is presented in Appendix A. The SI can be described as follows:

> Given a set of n ordered pairs $\{(a_i, b_i) \in \mathbb{N}^2 \mid a_i < b_i \ \forall i\}$, is there an integer value X such that the following equation is true for all $i \in [1, n]$?
>
> $$\forall i,\ X \not\equiv a_i \mod b_i \tag{3.3}$$

Indeed, it is equivalent to a reduced version of the problem approached in this chapter. In order to achieve the goal of data freshness, any proposed solution must at least be able to find a set of offsets in which there is no simultaneous release of any two threads (case where $\Delta_{i,j} = 0$). Thus, by constructing the solution one thread at a time (calculating the offset O_{i+1} of each new thread τ_{i+1} given that other offsets have already been defined), this algorithm would be effectively solving SI, with $X = O_{i+1}$, $a_i = O_i$ and $b_i = \gcd(T_i, T_{i+1})$.

Consequently, we suppose the problem of finding a set of offsets is at least as hard as SI. As SI is NP-complete, we consider our problem as being NP-hard. Hence, we have no hope of finding a polynomial algorithm that is able to solve it, as most probably $P \neq NP$. For that reason, heuristic approaches are the best alternative for trying to find a solution in a reasonable time.

The heuristics available in the state of the art rely on random or arbitrary values to calculate their propositions, or would require the redevelopment of schedulers to allow per-job offsets [NDB18]. On the other hand, optimization tools can be used to search for the best offset assignment given a set of constraints (thread periods and mathematical relations defining the scheduler characteristics) and a goal (no-interference state, or minimal delay). Yet, it must be noted that such methods explore the whole solution-space – every possible non-equivalent offset assignment – before a final solution can be proposed. Some specific methods allow the simplification of such spaces, but optimization tools still need to spend an exponential amount of time to propose a solution to a system with n threads. Even worse: most methods used nowadays tend to simulate every offset assignment proposition during the whole system's LCM before they can evaluate the quality of that proposition, and this process itself is exponential with regard to the input size n.

With a better understanding of the mathematical properties of offset free systems, a more tailored approach can be used, as described in the next section.

3.2 The Algorithm GCD+

We propose GCD+[Lad+22] as a new heuristic method based on an extrapolation, for sets of several threads, of the properties derived from sets of only two threads.

For any two-thread system containing τ_1 and τ_2, Equation 2.2 can be used to guarantee that no delay will ever happen, and the response time of each thread will be equivalent to its execution time. Notably, if $C_1 \leq \Delta_{1,2}$ and $C_2 \leq \Delta_{2,1}$, it is guaranteed that every job of τ_1 will always have finished its execution before any job of τ_2 is released, and vice-versa.

Overlap

Definition 3.1. Two threads τ_i and τ_j are said to overlap when either $C_i > \Delta_{i,j}$ or $C_j > \Delta_{j,i}$. In the former case, τ_i is said to overlap into τ_j of $(C_i - \Delta_{i,j})$ time units, and in the latter, τ_j overlaps into τ_i of $(C_j - \Delta_{j,i})$ time units.

In systems of n threads, if every possible thread pair respects this condition, then every job of every thread will also have a response time equivalent to its execution time, with no delays. This state of zero overlap is what GCD+ tries to reach. With that goal, GCD+ relies on the following concepts.

Definition 3.2.1 (The overall GCD, Ω)**.** The overall GCD, also referred to as Ω, is the greatest common divisor of all the periods (Equation 3.4).

$$\Omega = \gcd\left(T_i \ \forall i\right) \tag{3.4}$$

GCD+ is limited to sets of threads that present execution times at most as large as Ω. Otherwise, it will not provide interesting results. This hypothesis is often verified in real systems due to the human choices for the values of periods (greater tendency to using round numbers). Also, several methods have been proposed to lightly adjust existing thread periods into values that provide a small hyperperiod and large Ω [BC11]; [Xu10]; [Bro+11]; [RBR12]; [NF15].

Definition 3.2.2 (Cycle)**.** A cycle is a time interval of size Ω. The cycle k (for $k \in \mathbb{N}$) is the time interval $[k\,\Omega, (k+1)\,\Omega)$.

We can therefore see the execution of the system over time in a series of cycles, from cycle zero to infinity. Instead of a straight line, the passage of time can be seen as an infinite coil, with each loop containing Ω units of time, which will be useful for the algorithm. A simplified visualization of such a spiral is represented in Figure 3.2, where the linear passage of time is represented on top and a sequence of cycles is represented on the bottom. For this example, $T_i = 14$, $C_i = 2$, $T_j = 21$, $C_j = 1$, $O_i = 1$, $O_j = 4$ and $\Omega = 7$.

From definition 3.2.2, we can conclude that an offset O_i assigned to any thread τ_i can be related to a specific cycle O_{C_i} according to Equation 3.5. Conversely, in order to choose the

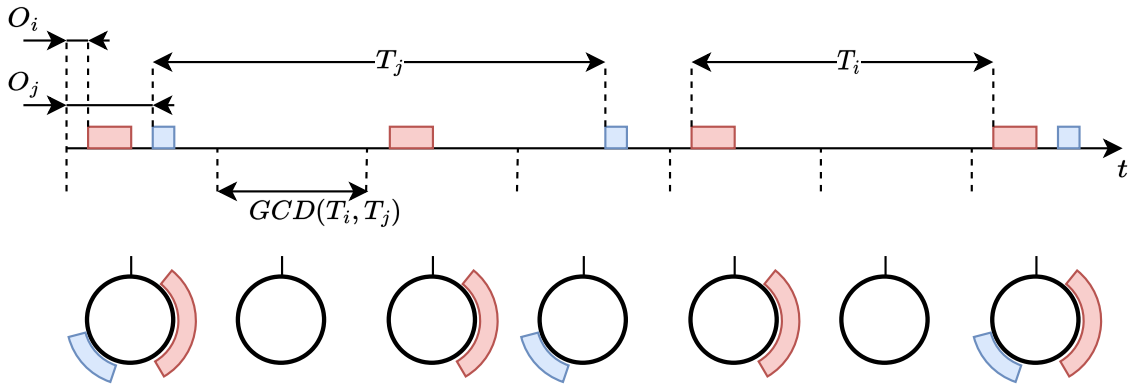

Figure 3.2: Visualization of time as a sequence of cycles

offset for a thread, we can at first choose its cycle, and then a residual offset to compose the final choice.

$$O_{C_i} = \left\lfloor \frac{O_i}{\Omega} \right\rfloor \tag{3.5}$$

Definition 3.2.3 (Subperiod, T_S)**.** The subperiod T_{S_i} of a thread τ_i is the ratio between its period T_i and Ω (Equation 3.6).

$$T_{S_i} = \frac{T_i}{\Omega} \tag{3.6}$$

The subperiod indicates the interval, in cycles, between two job releases of the same thread, i.e., any thread τ_i will be released once at every T_{S_i} cycles. From the definition of Ω, we know that $T_{S_i} \in \mathbb{N}\ \forall i$.

Lemma 3.2.1
For any given pair of threads τ_1 and τ_2, if $\gcd(T_{S_1}, T_{S_2}) = 1$, they must be released in the same cycle at some point in the hyperperiod of the system, no matter their offsets. Otherwise (i.e., if $\gcd(T_{S_1}, T_{S_2}) > 1$), τ_1 and τ_2 can be given offsets such that they will never be released in the same cycle.

Proof. This proof is related to the Generalized Chinese Remainder Theorem (GCRT). Let $\{T_i \in \mathbb{N} \mid i \in [1, n]\}$ and P be the least common multiple of all T_i. Let $a, O_1, O_2, \ldots, O_n$ be any integers. If $O_i \equiv O_j \mod \gcd(T_i, T_j)$ for $1 \leq i < j \leq n$, the GCRT states that there is exactly one integer k that satisfies the conditions:

$$a \leq k < (a + P), k \equiv O_i \mod T_i\ \forall i \in [1, n]$$

Otherwise (i.e., if $\exists (x, y) \mid O_x \not\equiv O_y \mod \gcd(T_x, T_y)$), then no such integer exists.

Thus, if $\gcd(T_{S_1}, T_{S_2}) = 1$ (coprime subperiods), regardless of the first cycle (O_{C_1} and O_{C_2}) where they are released, $O_{C_1} \equiv O_{C_2} \equiv 0 \mod 1$, and there must exist an integer $0 \leq k < \operatorname{lcm}(T_{S_1}, T_{S_2})$ such that $k \equiv O_{C_1} \mod T_{S_1}$ and $k \equiv O_{C_2} \mod T_{S_2}$. Then, in the cycle k, both threads will be released.

Otherwise, if $\gcd(T_{S_1}, T_{S_2}) > 1$, O_{C_1} and O_{C_2} might be chosen so that $O_{C_1} \not\equiv O_{C_2} \mod \gcd(T_1, T_2)$, making it impossible for such k to exist. In this case, τ_1 and τ_2 will never share the same cycle. □

Based on lemma 3.2.1, if we visually superpose every cycle into a single one, pairs of threads that seem to overlap and have coprime subperiods will certainly interfere with each other, generating a delay (since they will certainly share a cycle). However, threads with non-coprime subperiods can seem to overlap without actually interfering (case where they never share a cycle). This means that, from that perspective, while threads with coprime subperiods need to have distinct portions of the cycle reserved to them so they do not overlap, threads with non-coprime subperiods can share the same portion. The concept of a section can then emerge, grouping threads with non-coprime subperiods. A visual representation is presented in Figure 3.3.

Definition 3.2.4 (Section, Ψ)**.** A section Ψ is a subset of threads such that their subperiods are either all equal to one, or they must share a common prime factor. Thus, each section is related to a number $p \in \mathbb{P} \cup \{1\}$, where $\mathbb{P}$ is the set of all prime numbers. Each section Ψ_p is given a start time O_S and has a finite size δ, given by Equation 3.7.

$$\begin{aligned} \Psi_p.\delta = &\max_{\forall i | \tau_i \in \Psi_p} ((O_i + C_i) \mod \Omega) \\ &- \min_{\forall i | \tau_i \in \Psi_p} (O_i \mod \Omega) \end{aligned} \tag{3.7}$$

The dot notation is used to represent the property of a section: $\Psi.\delta$ is the size of the section Ψ, and $\Psi.O_S$ is its starting time. Then, for every cycle k, the interval $[k\,\Omega + \Psi_p.O_S, k\,\Omega + \Psi_p.O_S + \Psi_p.\delta)$ is reserved to the execution of threads in the section Ψ_p. In other words, every job of a thread in Ψ_p is only released at an instant r_{ij} such that $r_{ij} \geq k\,\Omega + \Psi_p.O_S$ and $r_{ij} + C_i \leq k\,\Omega + \Psi_p.O_S + \Psi_p.\delta$.

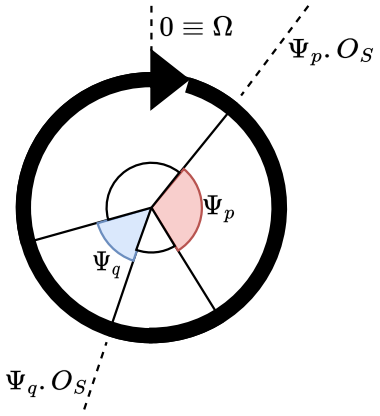

Figure 3.3: Representations of sections

For every section, O_S is assigned as a partial offset to every thread in the section, composing with O_C part of the final offset. Inside each section, however, some threads might need to

occupy the same cycle as well. In these cases, one of the threads has to be set to be released only after the other finishes its execution. For this, an internal offset O_I needs to be added in order to avoid overlapping. Hence, the final offset can be obtained from Equation 3.8.

$$O_i = \Omega\, O_{C_i} + O_{S_i} + O_{I_i} \tag{3.8}$$

In summary, GCD+ needs to:

1. Calculate Ω and, for every thread τ_i, its subperiod T_{S_i};
2. For each thread, choose the section in which it will belong, trying to minimize the total sizes of sections;
3. For each thread, choose in which cycle O_C it will be released first, setting an internal offset O_I in case it is needed;
4. Calculate each section size, and then each section offset O_C;
5. Calculate each final offset according to Equation 3.8.

A part of the initial problem complexity relies on in the partitioning subproblems of step 2 and step 3, that can be solved heuristically with usual methods such as a Best-Fit Decreasing algorithm, for example.

To better understand the aforementioned concepts, let us analyse the four-thread system containing the following:

$$\tau_1' = (T_1 = 24, C_1 = 2)$$
$$\tau_2' = (T_2 = 16, C_2 = 1)$$
$$\tau_3' = (T_3 = 16, C_3 = 3)$$
$$\tau_4' = (T_4 = 16, C_4 = 3)$$

Following the steps of the algorithm, first let us calculate Ω and each T_{S_i}. In this case, $\Omega = \gcd(24, 16, 16, 16) = 8$. Therefore, $T_{S_1} = 3$ and $T_{S_2} = T_{S_3} = T_{S_4} = 2$.

Now, based on each T_{S_i}, we need to choose a section for each thread. For this example, only two sections can exist: one for subperiods multiple of $p = 2$ (Ψ_2) and one for $p = 3$ (Ψ_3). Ψ_3 only contains τ_1, and the other threads are contained in Ψ_2. Since each thread has a single option for a section, the complexity of step 2 is avoided.

Then, for each section, the cycles can be chosen. The thread τ_1 can be released in any cycle without any difference, since it is the only thread in the section. The cycle 0 is arbitrarily chosen, and section Ψ_3 is done. Using a Best-Fit Decreasing heuristic, the thread with largest C is the first to be processed, i.e., either τ_3 or τ_4. Then, τ_3 can be assigned to cycle 0. Afterwards, τ_4 is assigned to cycle 1, and τ_2 can finally be assigned to cycle 0, since both its options are equivalent in terms of workload ($C_3 = C_4$).

However, τ_2 and τ_3 are in the same section and share the same cycle. τ_2 needs to have an internal offset $O_{I_2} = C_3$. Every other internal offset is zero.

Finally, calculating the section sizes is now possible. The size of Ψ_3 is $C_1 = 2$, and the size of Ψ_2 is $\max(O_{I_i} + C_i)\ \forall \tau_i \in \Psi_3$, which is $C_3 + C_2 = 4$. We can then choose $\Psi_2.O_S$ to be 0, and then $\Psi_3.O_S = \Psi_2.\delta = 4$.

The calculated parameters are shown in Table 3.1. Hence, every offset is calculated according to Equation 3.8, and the resultant system is shown in Equation 3.9. The threads are represented in the modular circle of size Ω in Figure 3.4, and their schedule is represented over the whole LCM in Figure 3.5.

Table 3.1: Calculated values for each thread in the example

Thread	T_i	C_i	p	O_{C_i}	O_{I_i}	$\Psi_p.\delta$	O_{S_i}
τ_1	24	2	3	0	0	2	4
τ_2	16	1	2	0	3	4	0
τ_3	16	3	2	0	0	4	0
τ_4	16	3	2	1	0	4	0

$$\begin{aligned}
\tau_1 &= (T_1 = 24, C_1 = 2, O_1 = 4) \\
\tau_2 &= (T_2 = 16, C_2 = 1, O_2 = 3) \\
\tau_3 &= (T_3 = 16, C_3 = 3, O_3 = 0) \\
\tau_4 &= (T_4 = 16, C_4 = 3, O_4 = 8)
\end{aligned} \tag{3.9}$$

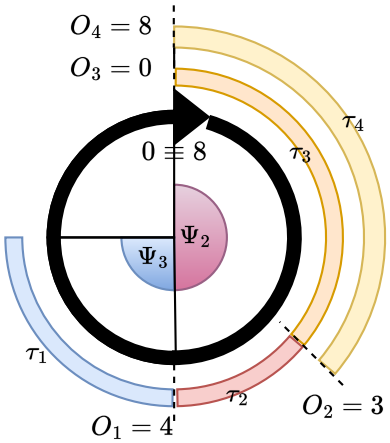

Figure 3.4: Cycle representation of the scheduled threads in the current example

Lemma 3.2.2
If $C_i \leq \Omega\ \forall i$ and $\sum \Psi_p.\delta \leq \Omega\ \forall p$ (considering $\Psi_p.\delta = 0$ for empty sections), then the system presents zero interference, i.e., every thread's response time equals its execution time C_i.

Proof. The proof relies on analysing each possible thread pair in a system, proving they will

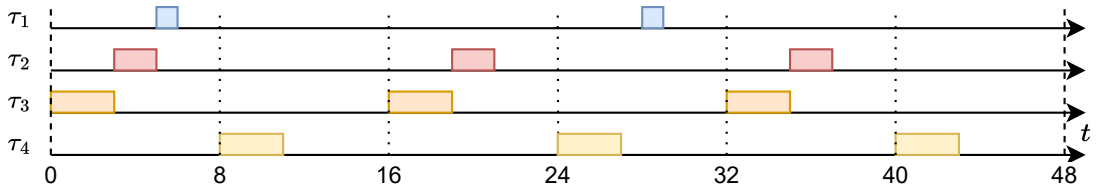

Figure 3.5: LCM representation of the scheduled threads in the current example

never overlap. Then, by construction, the whole system of n threads cannot present any execution delay.

For any thread τ_i, there are three possibilities regarding the analysis of a second thread τ_j:

1. τ_j is assigned to a different section than τ_i;
2. τ_j is assigned to the same section but in a non-congruent cycle than τ_i;
3. τ_j is assigned to the same section and a congruent cycle to τ_i.

In the first case, from the definition and properties of sections, if the sum of section sizes is at most equal to Ω, then threads in different sections can never overlap.

For the second case, knowing that $\sum \Psi_p.\delta \leq \Omega$, that every thread execution is contained in the limits of its section, and that τ_i and τ_j do not execute in the same cycle, then each thread will always be executed inside a different cycle. Therefore, every job of τ_i will have finished its execution before τ_j starts, and vice-versa.

For the third case, thanks to the internal offset O_I and knowing that a section size is not greater than Ω, then every job of τ_i will have finished its execution before τ_j starts, and vice-versa. □

As a consequence of lemma 3.2.2, GCD+ needs to make its choices seeking to reduce the sum of section sizes. Therefore, when adding a new thread τ_i with subperiod T_{S_i}, if more than one prime number composes T_{S_i}, GCD+ will test every possibility so that the section with the least increase in its duration is chosen.

For each tested section, in order to choose the cycles, threads are analysed in a decreasing order according to their execution times C_i. Being τ_i the thread currently in analysis, a vector of size T_{S_i} can be made, and for every thread τ_j in the same section with an already assigned cycle, the vector is filled with the expected response time value $O_{I_j} + C_j$ at every position $x \equiv O_{C_j} \mod \gcd(T_{S_i}, T_{S_j})$. The cycle with the smallest value (ideally, zero) is chosen for τ_i, and the value of the internal offset O_{I_i} is the value that was registered in the chosen cycle.

Using Euclid's algorithm to find GCDs in $\mathcal{O}(\log T_{max})$, a trial division to find the prime factors of each period in approximately $\mathcal{O}(\sqrt{T_{max}})$, and considering the number of distinct prime factors of any integer to be at most $\frac{\log T_{max}}{\log \log T_{max}}$, the time complexity of GCD+ is then

$\mathcal{O}(n \cdot (\log T_{max} + \sqrt{T_{max}} + \frac{\log T_{max}}{\log \log T_{max}} \cdot (\log T_{max} + T_{max})))$. Note that the last T_{max} is only a superior bound to T_{S_i}, and the algorithm generally works very far from this bound.

Algorithm 1 GCD+

Require: $\Phi = \{(T_i, C_i) \mid i, T_i, C_i \in \mathbb{N} \wedge i \in [1, n]\}$
Ensure: $O = \{O_i \mid i, O_i \in \mathbb{N} \wedge i \in [1, n] \wedge O_i < T_i\}$

Reorder threads decreasingly w.r.t. C_i
$\Omega \leftarrow \text{GCD}(\{\forall\ T_i\})$
for all $\tau_i \in \Phi$ **do**
    if $T_i = \Omega$ **then** assign$(\tau_i, 1)$
    else
        Calculate T_{S_i} (Equation 3.6) and find its prime factors
        for all $p \in$ `primeFactors`(T_{S_i}) **do**
            Construct vector v of size T_{S_i}, with $v_k = 0\ \forall k$
            for all $\tau_j \in \Psi_p$ **do**
                $x \leftarrow O_{I_j} + C_j$
                for all $k \leq T_{S_i} \wedge k \equiv O_{C_j} \mod \gcd(T_{S_i}, T_{S_j})$ **do**
                    if $x > v_k$ **then** $v_k \leftarrow x$
                    end if
                end for
            end for
        end for
        Choose p with minimal v_k (ideally zero)
        $O_{C_i} \leftarrow k$ (the index of the minimal v_k)
        $O_{I_i} \leftarrow v_k$
    end if
end for
for all p **do**
    Section size $\Psi_p.\delta \leftarrow \max(O_{I_j} + C_j\ \forall \tau_j \in \Psi_p)$
    $\Psi_p.O_S \leftarrow$ previous section size
    for all $\tau_i \in \Psi_p$ **do**
        $O_{S_i} \leftarrow \Psi_p.O_S$
    end for
end for
for all $\tau_i \in \Phi$ **do** calculate O_i according to Equation 3.8
end for

To summarize, GCD+ divides the problem of finding offsets into smaller problems. Although the sub-problems can still be NP-hard, such as choosing the sections or the cycles (similar to bin packing problems), these have much smaller inputs and fewer possible outputs than the original problem (Equation 2.3).

3.3 Experiments

To evaluate the efficacy of GCD+, two tests were made. First, randomly generated sets were used as inputs to GCD+ and other heuristic methods, and their results were compared. Then, a real case study in Paparazzi was used, where independent periodic threads were represented by telemetry messages in a single communication link, which had message losses due to excessive simultaneous messages being put in the sending queue.

3.3.1 Randomly Generated Sets

3.3.1.1 Methodology

Random sets of n threads are generated according to the methodology proposed in [GBD20]. For a given input integer n, this method generates n unbiased individual utilization factors u_i. The total utilization factor U is the sum of every individual utilization factor u_i.

Then, the individual utilization factors u_i are decomposed in their respective period T_i and execution time C_i. Knowing that $u_i = C_i/T_i$ and that the periods will define if the system is suitable for GCD+ or not, we focus on generating the periods, and the execution times will follow.

For generating representative thread periods, there are two available methods in the literature. The first one, used in [ASN16] for generating semi-harmonic thread sets, chooses a random integer between defined lower and upper bounds, and multiplies the random number to a given constant. Every period will therefore be a multiple of this constant. The second one [GM01], more general than the first, generates thread periods from a finite set of possibilities according to a given prime factor distribution. The latter was chosen since it is more general and representative of real-world scenarios.

In this method, a set of prime numbers $\{p_i\}$ is given to the algorithm as an input. Each prime p_i is related to a unique vector $\{x_j \in \mathbb{N}\}$ of size n_i. The vector represents the probability distribution that the corresponding exponent will be chosen to compose the final period. In other words, $x_0/\sum_k x_k$ is the probability that p_i will not divide the generated period, $x_1/\sum_k x_k$ is the probability that p_i will divide the generated period only once, $x_2/\sum_k x_k$ is the probability that p_i^2 will divide the generated period, and so on. The possible values for the periods are therefore limited to $\prod_i p_i^{(n_i-1)}$, and so is the hyperperiod of the system.

As a reference factor distribution, the periods of all sets of messages from Paparazzi[3] are analysed, and their factor distribution is obtained. We must recall that message periods from Paparazzi, in addition to being used in the telemetry system of operating drones, are tuned such that the hyperperiod is increased (using 11.1 seconds instead of 10, for example). This causes the overall GCD to be reduced, approaching then a worst-case scenario in our domain

[3] /paparazzi/conf/telemetry/default_rotorcraft.xml

of application.

Finally, the execution times are obtained from generated periods and individual utilization factors. They are rounded to the nearest integer, and thread sets that have any execution time rounded to 0 are discarded, since they will not have the effective number of threads that were demanded. Also, only sets for which Ω is greater or equal to the largest execution time will be used. This case is often found in real applications such as Paparazzi, and is needed for our heuristics to work properly.

The algorithms for finding the offsets can then be used in each thread set. The compared heuristics are:

Goossens's Heuristic [Goo03] (original and modified): a list of every possible pair of threads is constructed and, from the largest GCD of the threads' periods, offsets are defined such that $O_j = O_i + \frac{1}{2}\text{GCD}$. If O_i is not yet defined, a random value is assigned. The modification consists of defining $O_j = O_i + \frac{1}{2}(\text{GCD} + C_i - C_j)$.

CAN Message Heuristic [GHN08] : thread releases are put in the middle of the largest empty interval in the interval $[0, T_{max}]$.

Paparazzi's Heuristic [Hat+22]: the offset for the thread τ_i is calculated according to the expression: $O_i = [(i-1)\text{mod}10] \cdot T_i/10$.

GCD+ : as defined in the previous section.

After each offset assignment is obtained, the respective concrete thread sets are simulated over an interval equivalent to two hyperperiods plus the greatest offset, as that interval will certainly contain the infinitely repeating cycle of the system [CGG04]. During the simulation, for each thread in each set, its maximum delay η_i is registered, being η_i the largest time interval between a job activation (thread release) and the beginning of its respective execution.

The delays are then put directly in a box plot to be analysed in their brute form. Each heuristic will have a box plot containing the maximum delays of every thread in every thread set for a given total utilization factor.

However, while a certain absolute value for a delay (for example, 1000 time units) can mean a significant deadline miss (e.g., a thread with period 500 considering implicit deadlines), for others it can mean only an "affordable" amount of 10% of its period (e.g., a thread with period 10000 in a system with few and low-utilization threads). Therefore, the normalization of each delay with regard to its thread period is required for it to reflect its impact on the thread system.

The choice of a normalization eases the interpretation of the delays. For example, if a thread was delayed at most by 100 time units in a system under FIFO in which the thread with maximum WCET executes in 50 time units, more threads were in the queue at the moment of its release than in another system where we observe the same maximum delay, but with a maximum WCET of 200 time units. In the former, thread delays have been chained and, in the case of communication links, more messages are kept waiting. Hence, to look for signs of chaining delays, we normalize the delay over the maximum execution time of other

threads in the set.

In addition, to evaluate the response time of each thread, the sum of the maximum delay and the execution time of each thread is normalized with respect to each execution time. Values close to 1 will indicate a small relative change to the response time.

A final analysis takes into account the schedulability of each thread set under FIFO. If any implicit deadline was missed, the thread set is marked as unschedulable under FIFO. Then, the amount of schedulable sets is counted for each offset-assignment method, and they are compared between each other for each value of the total utilization factor.

In summary, the metrics used in the analysis are in the following list:

1. brute maximum delay η_i
2. maximum delay normalized by period η_i/T_i
3. maximum delay normalized by the maximal execution time of other threads, i.e., $\eta_i/\max_{\{\forall \tau_k \neq \tau_i\}}(C_k)$
4. maximum response time $(\eta_i + C_i)/C_i$
5. schedulability

3.3.1.2 Results

The algorithms were implemented and executed in Python 3.9.6, using a laptop with Ubuntu 18.04.1, Intel Core i7-4710MQ CPU (2.50GHz, 4 cores, 8 threads, 2054MHz), and 16 GB of RAM.

Sets of 8 and 16 threads are used as input systems. Due to the duration of the simulations, no larger sets were used[4]. The time for each method to yield an offset assignment is measured and, later, the resulting offset assignments are evaluated in a simulation according to the criteria presented before. Results were generated using 1000 sets and 2000 sets. There was no noticeable difference between graphs of both cases, so we consider 1000 sets to be sufficient for an experiment to be representative.

The offset calculation time for each group of sets is shown in Table 3.2 for sets with 70% utilization. Other utilization values did not present significant relative variations from the values of the table.

Table 3.2: Average time to assign offsets (results from 1000 filtered sets at 70% utilization)

Method	Time - 8 threads (µs)	Time - 16 threads (µs)
GCD+	54.8	151
Paparazzi method [Pap]	4.78	9.11
Goossens's method [Goo03]	22.8	74.7
CAN message method [GHN08]	45.7	346

[4]Note that this limitation does not relate to the limits of the proposed heuristics, but to the time it takes to evaluate their results.

Table 3.2 shows that the time to calculate offsets is very small from a human perspective for every method. It also indicates that every method is scalable, and therefore we can focus solely on their results.

Box-plots are used to represent simulation results, where each heuristic method has its box. The mean value is also represented, as a dashed green bar. In the plots, the median (50^{th} percentile) is represented as an orange continuous bar. The limits of the box are the first quartile (25^{th} percentile) and third quartile (75^{th} percentile). Whiskers span from a box limit until the furthest value such that its size is, at most, 1.5 times the distance between the first and third quartile. All values beyond the whisker limits are considered outliers and are not represented, in order to allow for readability. However, the representations showing outliers follow the same tendencies as shown in Figures 3.6 to 3.9.

For sets of 8 threads, the behaviour of the systems can be seen in Figure 3.6 for 70% processor utilization and in Figure 3.7 for 90% utilization. Similarly, the behaviour for 16 thread systems is seen in Figure 3.8 for 70% utilization and in Figure 3.9 for 90% utilization. In these figures, the box plot furthest to the left shows the raw maximum delays (η_i), the one at its right side shows the normalization η_i/T_i, the third box plot shows the normalization by the maximum execution time, and the fourth and last box plot shows the maximum response time normalized by the thread's execution time.

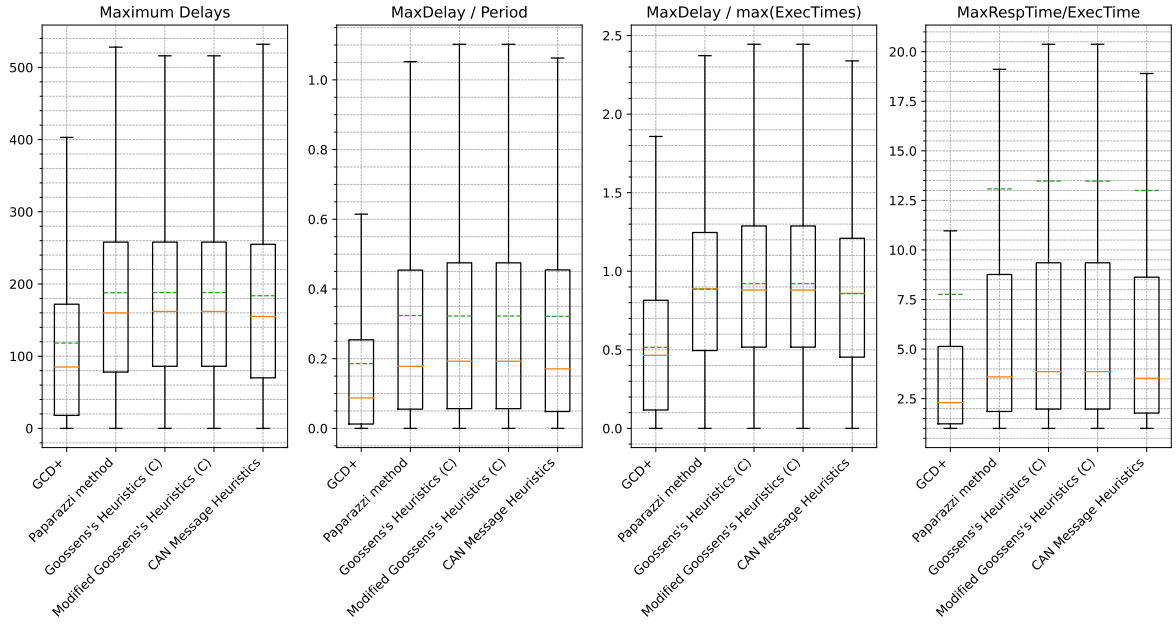

Figure 3.6: Delays extracted from simulations for 1000 filtered sets of 8 threads, with $U = 70\%$

From the box plots, it can be seen that every quartile (first, second and third) relative to GCD+ has significantly smaller values in every situation than the ones relative to the other heuristics, as well as the boundaries of the whiskers and the mean values. For utilization factors of 70%, GCD+ was able to keep the delay of the great majority of threads under 60% and 80% of their periods in the case of sets of 8 and 16 threads, respectively. Other heuristics,

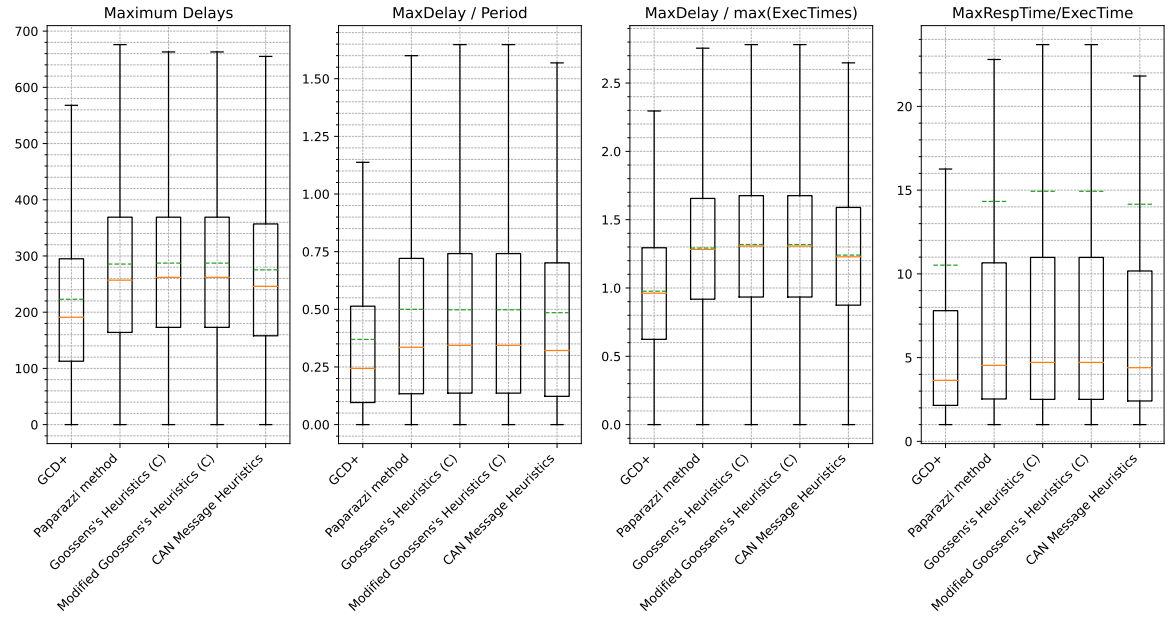

Figure 3.7: Delays extracted from simulations for 1000 filtered sets of 8 threads, with $U = 95\%$

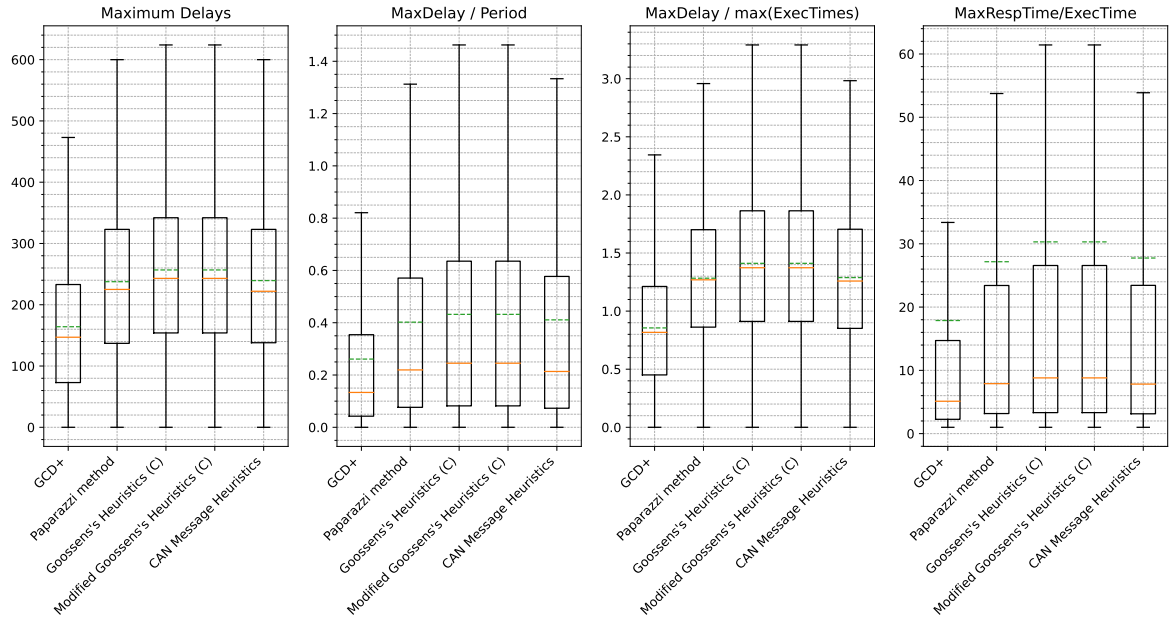

Figure 3.8: Delays extracted from simulations for 1000 filtered sets of 16 threads, with $U = 70\%$

however, show whiskers going beyond the value of 100% in these same situations.

It is also noticeable how every heuristic method has more trouble reducing delays when the thread count increases, notably with respect to the ratio between maximum response times and execution times – this ratio tripled when the thread count doubled. Increasing

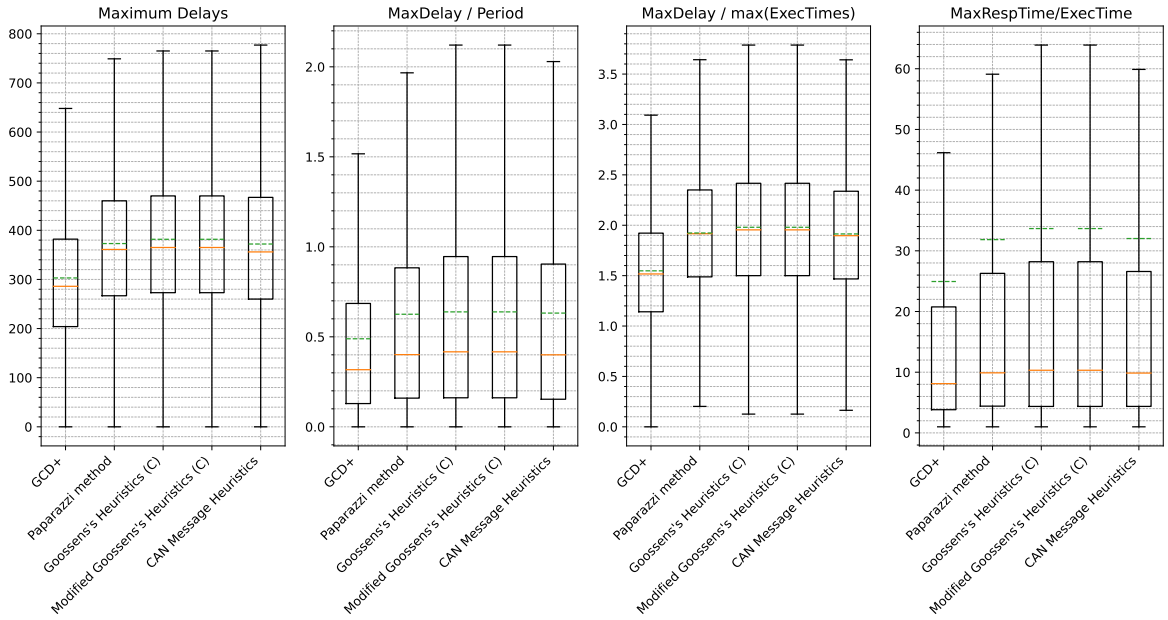

Figure 3.9: Delays extracted from simulations for 1000 filtered sets of 16 threads, with $U = 95\%$

the utilization factor is another aspect that worsens the maximum delays for every heuristic method.

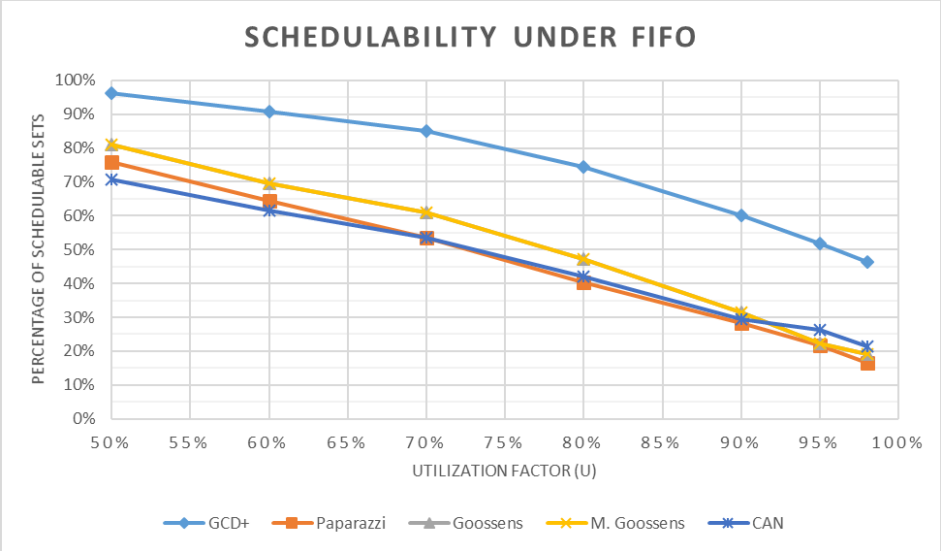

Figure 3.10: Schedulability for 1000 filtered sets of 8 threads, from $U = 50\%$ to 98%

In order to evaluate the schedulability of these thread sets with respect to the utilization factors, an implicit deadline was considered for every thread, and if a set has a thread that missed a deadline, it is considered unschedulable. Figure 3.10 and Figure 3.11 were then obtained for sets of 8 and 16 threads, respectively. GCD+ shows a significant advantage

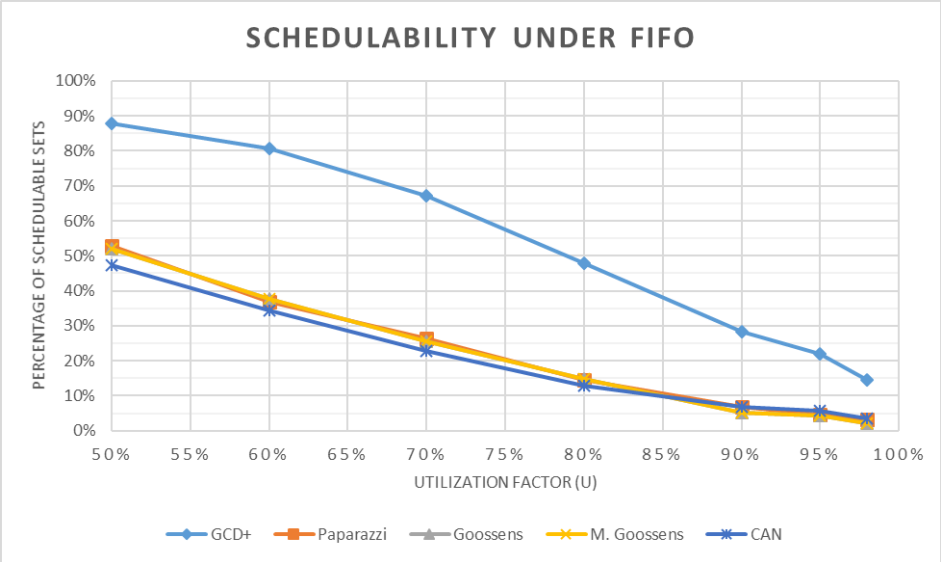

Figure 3.11: Schedulability for 1000 filtered sets of 16 threads, from $U = 50\%$ to 98%

relative to every other heuristic in every situation that is shown. For sets of 8 threads, at 80% utilization factor, there is an increase of at least 50% in the number of schedulable sets, while for 90% utilization there are about twice as much schedulable sets when using GCD+. For sets of 16 threads, the increase is even more pronounced: from about 2x at 50% utilization to about 4x at 90%.

The significant improvement seen in the presented results can be explained by certain limitations of the other methods. Paparazzi method tries to distribute the offsets without taking into account the relations between message parameters, but instead using an arbitrary rule that has no theoretical support. The CAN Message heuristic, when trying to distribute the offsets over the maximum period, ignores the effects over the hyperperiod, during which two threads that were initially put far away can coincide. The Goossens's heuristic method, although it considers some interactions between pairs of threads during the whole hyperperiod, has a trap: when several threads have the same period, it will tend to assign the same offset to all of them, unintentionally creating a critical case. Furthermore, its random component might lead to undesired critical cases. The fact that it does not take into account the execution times of threads has no apparent effect in its results, since the Modified Goossens's Heuristic (which does take them into account) had very similar results to the original method.

In addition, tests were made to consider optimization tools such as CPlex and ORTools. Nonetheless, the time it takes to calculate a single offset assignment with the available resources was incompatible with the need to evaluate a thousand different sets, and most attempts using sets with more than nine threads could not propose any solution even after days of continuous calculations.

3.3.2 Case Study: Paparazzi telemetry

An adapted real-world scenario was set up using the telemetry of Paparazzi. A telemetry configuration file was used to define the messages that a Paparazzi autopilot implementation sends periodically, such that the telemetry is composed of the messages described in Table 3.3. The "Size" column does not consider any protocol header or tail.

Table 3.3: Paparazzi telemetry values

i	ID	Period (s)	Size (B)
1	ALIVE	2	17
2	ROTORCRAFT_FP	1	58
3	INS_REF	1	32
4	ROTORCRAFT_NAV_STATUS	1	15
5	ENERGY	1	21
6	DATALINK_REPORT	1	11
7	DL_VALUE	0.2	5
8	ROTORCRAFT_STATUS	0.2	20
9	STATE_FILTER_STATUS	0.2	4
10	AIR_DATA	0.2	28
11	INS	0.2	36
12	GPS_INT	0.1	57
13	IMU_GYRO_SCALED	0.04	12
14	IMU_ACCEL_SCALED	0.04	12
15	IMU_ACCEL_RAW	0.02	12
16	IMU_GYRO_RAW	0.02	12

The messages are sent to a UART channel, being normally connected to an antenna to transmit the data. However, for this test and in order to ignore any effects related to the transmission of radio waves, messages are read directly in the UART channel. A representation of this setup is presented in Figure 3.12.

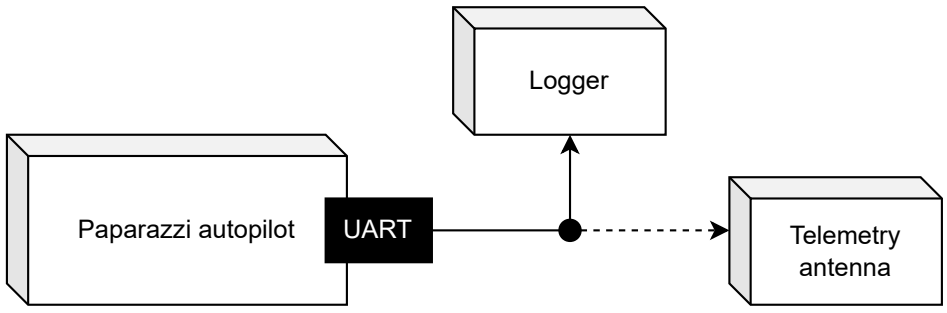

Figure 3.12: Representation of the physical experiment setup of the case study

According to the UART protocol, every byte has a start and an end bit added to it, which makes it transmit 10 bits per byte. The channel was configured to transmit 57600 bits per second. In addition, the Paparazzi protocol used (Pprzlink V2.0) adds 8 bytes to each

message for header and checksum[5].

Therefore, we can use the time it takes for a bit to be transmitted as a unit of time, i.e., 1/57600 second, allowing the system to be described only by integers. For the conversion of units, the values for periods need to be multiplied by 57600, and the amount of bytes in every message, after the 8 Pprzlink bytes are added, have to be multiplied by 10.

With these converted values, we can generate offsets using GCD+ and the other heuristics. Later, a simulation can be run in order to evaluate the delays related to each set of offsets. The resulting delays are presented in Figure 3.13, this time showing also the outliers.

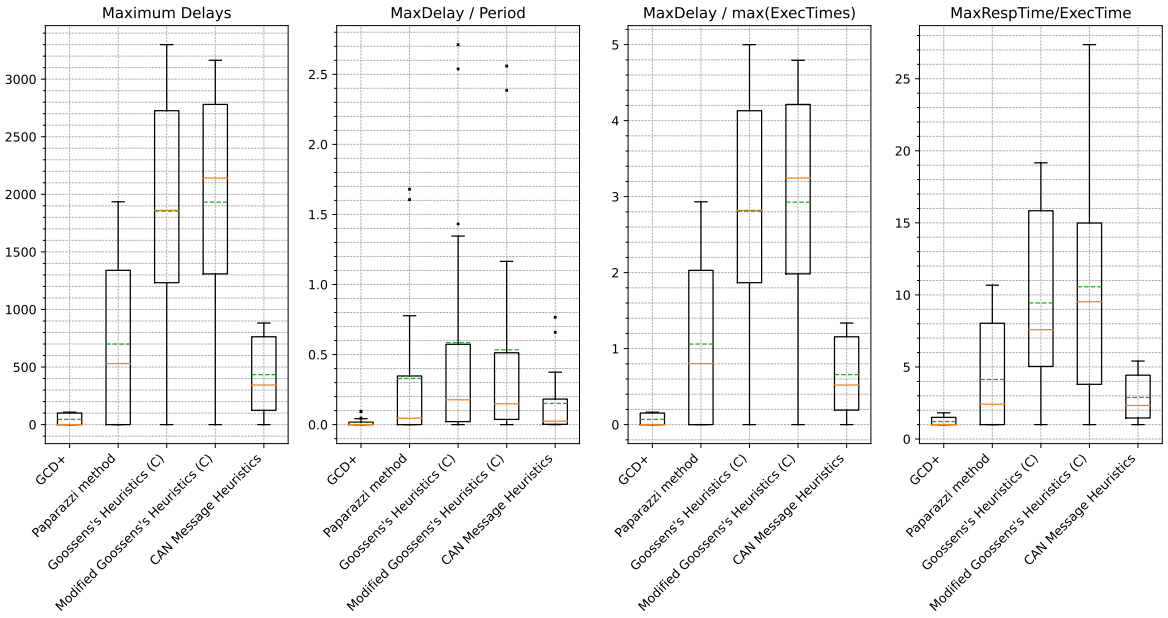

Figure 3.13: Delays extracted from simulations for the case study

Figure 3.13 shows that, using GCD+, every message has a delay under 10% of its period, in comparison to 80%, 170% and even 270% in other heuristics. These delays represent at most 20% of the maximum length of other messages, while the third metric shows that there was a significant chaining of interferences for all the other heuristics. Also, deadline misses were registered for the methods Goossens and Paparazzi, while the system was schedulable using offsets from GCD+ and CAN Message heuristics.

From the physical analysis, before the implementation of GCD+, some messages were lost due to a full buffer, as it can be seen in Figure 3.14. The figure is a screen capture from the software Logic 2, for analysing the data acquired from the UART setup. The first line (Channel 0) represents the raw bits sent in the UART port; the second line (Channel 1) goes from a low to a high state every time the autopilot blocks the UART buffer, as it is writing a message to be sent in the channel, and from high to low when in releases the buffer; the third line (Channel 2) goes from high to low or from low to high when the buffer refused to accept

[5]wiki.paparazziuav.org/wiki/Messages_Format

a message from the autopilot due to it being already full (too many messages in the queue), i.e., a message loss. It can be seen that there is a message loss at every 2 seconds.

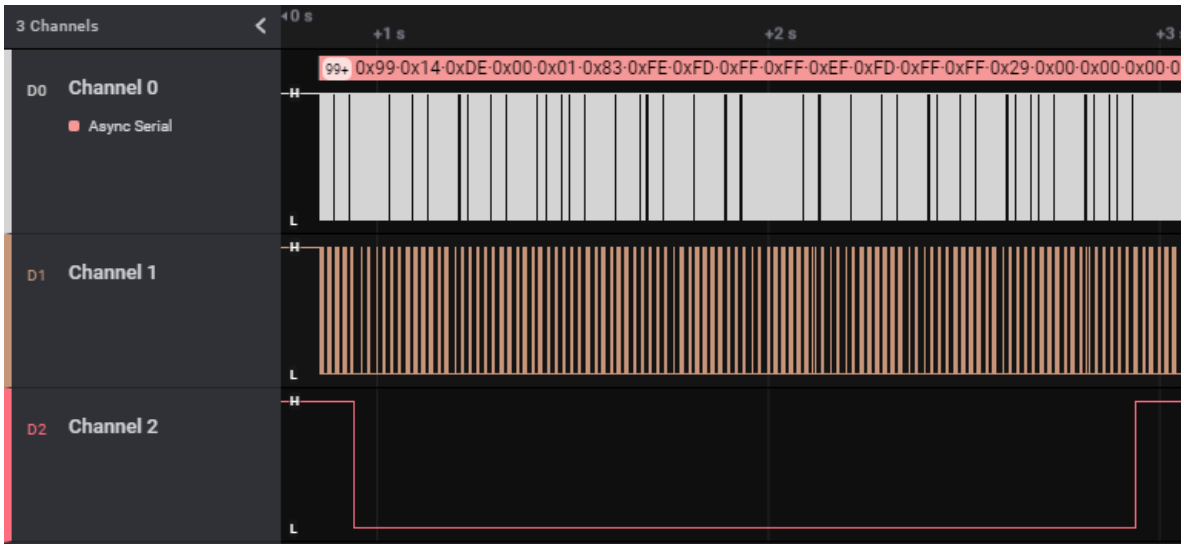

Figure 3.14: Screen capture from the analysis of a Paparazzi telemetry case with a message loss at every 2 seconds

Message losses were avoided with the offsets given by the heuristics presented in this paper, as it can be seen in Figure 3.15, where no message is ever lost. Both the screenshots show less than 3 seconds of data capture, but the complete test was made for 20 seconds.

Besides avoiding message losses, GCD+ also increases data freshness. As it can be seen in the second box plot of Figure 3.13, no message waits more than 10% of its period to be sent over the communication channel. This means that, for example, a watchdog could detect a communication loss much sooner than with other schedules. Also, control algorithms relying on this communication would have their stability margins increased.

The case study confirms the tendency seen in simulations: GCD+ vastly outperforms other heuristics.

3.4 Working with precedences

GCD+ is best used when each thread or message is independent (which is the case for the messages in a communication link such as in Paparazzi). If there exists any dependency between threads, for example the need for one of them to be executed after a given delay, this method will need to be adapted.

At first, thread precedence has to be well modelled. While precedence with distinct periods can be interpreted with the SPC approach [Ngu+18], for the sake of simplicity, we will only consider precedence relations between threads with identical periods. For those, the following

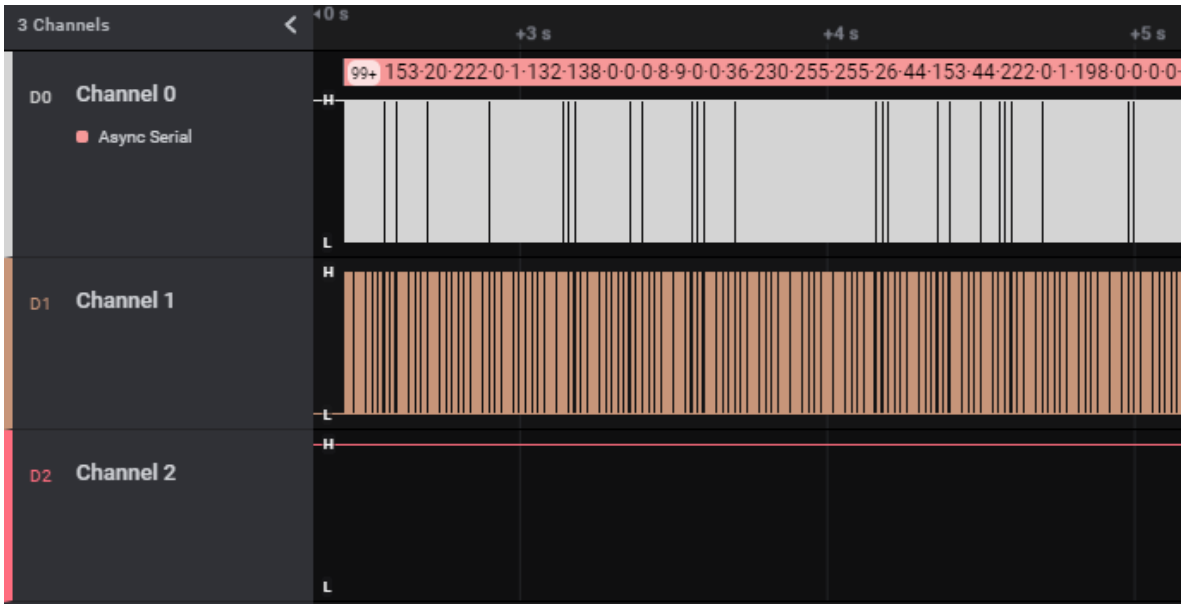

Figure 3.15: Screen capture from the analysis of an improved Paparazzi telemetry case with no more message losses

definition will be used.

thread Precedence

Definition 3.2. A precedence relation can be expressed between threads with identical periods as follows: $\tau_i \overset{X}{\prec} \tau_j$. The notation indicates that τ_i precedes τ_j of $X \geq 0$ time units. Therefore, each job of τ_j can only execute after X units of time have passed after the end of the respective job of τ_i. If $X = 0$, then τ_j should execute right after τ_i finishes.

Thread precedences therefore connect two threads, and multiple connections can form precedence networks. Each network will be in the form of a weighted Directed Acyclic Graph (DAG): precedences are weighted by X and directional, and no precedence cycle can occur, since it would result in a paradox (a thread would need to precede itself). DAGs have a topological order [Pan+15] forming an equivalent chain, and these chains can be used to improve choices in GCD+ to preserve the precedences.

Hence, an adaptation of GCD+ that would assess thread precedence would need to:

1. Calculate Ω and subperiods, and express each precedence weight X in terms of cycles (using Ω as a unit of time);
2. Start with threads in precedence chains (chained threads): choose their sections, and then the sections for threads outside any precedence chain (free threads);
3. For each section, starting with chained threads, choose the cycle (O_C) in which each thread will be released, respecting the precedence chain, and then do the same for free threads (setting an internal offset (O_I) in case needed;

4. Calculate section sizes and define each section offset (O_S);
5. Calculate each final offset according to Equation 3.8.

The third step is the core difference in the algorithm, since it needs to choose the cycles for several threads at once, aiming to minimize the workload in each cycle.

3.5 Conclusion

GCD+ has proven to be an efficient algorithm for scheduling independent threads in a non-preemptive scheduler, or messages in a single communication link. It is well suited for quasi-harmonic systems, which is common in real systems. For those, it is able to schedule sets better than the methods found in the literature, being even capable of a schedulability increase of 500% in some situations.

Its concepts can also be applied to threads in cyclic schedulers, since they have a defined cycle size and a non-preemptive behaviour. However, precedence relations distinguish real systems from the models of independent threads. Therefore, an adaptation to GCD+ to consider precedence chains was suggested.

With the recommended modifications, GCD+ can schedule threads and communication links in an embedded system. The offset-based schedule can be easily created and implemented in current autopilots, and it represents a further step into guaranteeing the safe operations of the drone.

Furthermore, GCD+ can be expanded to treat several communication links at once. With the proper adaptations, discussed in the following chapter, the algorithm can be used to schedule the frames in time-triggered switched networks such as TSN or TT-Ethernet.

Chapter 4

GCD#: Network Scheduling

In chapter 3, we have proposed GCD+, a scheduling heuristic that is very efficient in mitigating interferences in strictly periodic independent task systems executing in a single processor. A similar problem that requires mitigating interferences is the static scheduling of messages in a switched network. Although GCD+ can be used to schedule messages in a single communication link, if more links are connected forming a network, the method might no longer be applicable as-is. Yet, the analysis of some networks might be fundamental to guarantee the continuous execution of certain critical systems.

A widely used protocol for switched networks is the Ethernet protocol. Based on it, the TSN standards (incorporated in the standard IEEE 802.1Q) [Seo+21] provide several mechanisms that make the Ethernet protocol better adapted to time-critical requirements. Some examples of TSN mechanisms are Credit-Based Shapers, Time-Aware Shapers, Per-Stream Filtering Policies, etc.

In the core of some TSN mechanisms is the notion of time slots reserved to specific frames. With such strategies, issues such as bursts and high latency might be avoided, improving the "freshness" of consumed data. In order to define a proper time-slot schedule, most techniques used nowadays rely on constraint programming and optimization tools such as SMT and OMT (e.g., [RPC20]). In the hyperperiod, each frame of every flow is modelled as mathematical and logical constraints that need to be respected, and a solution space is explored such that these constraints are not violated. The solution space in case is the set of all possibilities of time slots. Given that each flow increases the solution space by at least one dimension, and that each dimension might hold thousands of possible choices for the beginning and the end of the time slot, it is clear that the solution space grows very fast for each new frame flow in the network. This explains why such an approach is time-consuming: a solution might need hours, even days of computational work to be found.

In contrast, the ideas that form the basis of GCD+ can be reused to quickly (pseudo-polynomial time) find a possible good candidate for an initial offset assignment, from which subsequent time slots can be easily calculated. In addition, its founding properties can be used to quickly (polynomial time, $\mathcal{O}(n^2)$) check if the configuration has reached a minimal latency condition, as explained in the following sections.

An adjustment to GCD+ is required to allow the scheduling of networks, and the goal of this chapter is to propose such an adaptation, called GCD#. This approach has proven to be sufficiently scalable to the analysed test cases.

4.1 Definitions and Considerations

A TSN is composed of individual systems interconnected by a switched Ethernet network. The individual systems, hereafter called "end systems", are connected to a single switch, while switches can be connected to several other switches and end systems. Switches have input ports and output ports, each port connected to a single end system or a single switch. An example is shown in Figure 4.1.

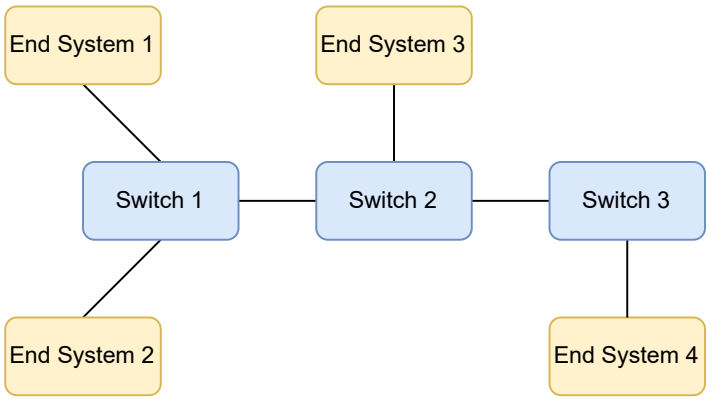

Figure 4.1: Example of end systems and switches connected in a representation of a physical network

Physical node E_i

Definition 4.1. A physical node E_i is an End-System or a Switch that belongs to a switched network.

End-System

Definition 4.2. An End-System is a physical node that can produce or consume messages in a network, but it cannot forward received messages.

Switch

Definition 4.3. A switch is a physical node that can only forward received messages from a communication link to another communication link.

Communication link $\overline{E_iE_j}$

Definition 4.4. A link defined by $\overline{E_iE_j}$ is a connection between the physical nodes E_i and E_j.

Note that each communication link is bidirectional, i.e., $\overline{E_iE_j} \equiv \overline{E_jE_i}$. For example, between the End System 1 and Switch 1, there is an output port of the End System 1 linked to an input port of Switch 1, and an output port of Switch 1 linked to an input port of End

System 1. Also, the network must operate in a defined baud rate, which expresses how much information can be transmitted per unit of time through the cables.

If communication was instantaneous, messages leaving a producer end-system would arrive at the same moment in the consumer end-system, needing only the time it takes to write them at a certain baud rate. However, in addition to the propagation time in the wires, each switch needs to completely read a message coming from each input port before forwarding them to the corresponding output port. Then, they are rewritten in the next communication link. This represents a technological delay that we define as follows.

Technological Delay (TD)

Definition 4.5. The technological delay (TD) is the amount of time a frame is delayed due to the switching technology. It is composed of the propagation time in the communication link (dependent on the link's physical length and signal speed), the time the switch needs to store the message in the input port (dependent on the message size and the link throughput), and the time it takes for the switch to forward the stored message from its input port to the corresponding output port (dependent on the technology).

For AFDX networks [Ari], for example, the technological delay is 16 microseconds.

End-to-end Delay (latency)

Definition 4.6. The end-to-end delay, also referred to as latency, measures the difference in time it takes for a bit of information to arrive at its final destination after it has left its producer end-system. It is composed of the technological delay plus any eventual periods of time these bits need to wait in output queues for other messages in the queue to be sent.

Smaller latencies are desirable in order to provide fresh data to the systems that receive them. Control systems, for example, are often more stable when their data is fresher. However, an equally important parameter to reduce is the jitter.

Jitter

Definition 4.7. The jitter is the possible variation in latency for a frame in a given topology and configuration.

A network of strictly periodic messages with zero jitter is deterministic: every message's time of emission and arrival is known, and their arrival is also strictly periodic, even if shifted in time by some latency. This characteristic allows the consumer end-systems to quickly identify message losses, since messages are expected to be received after one message period of the last reception. When message losses are quickly identified, the system can take the necessary measures to counteract the issues that could have caused it. For example, at the loss of a positioning system, if its system is proven to have low jitter, the drone can rapidly

engage in fail-safe mode instead of flying with incorrect positioning data.

Note that jitter is zero not only if latency is zero, but if latency is any constant value. Therefore, we are interested in reducing latency to a minimal constant value.

Minimal Latency Condition

Definition 4.8. The latency of any frame can be calculated by the sum of: 1) its transmission and switching time for its whole path; and 2) the time it has to wait in any queue for the transmission of other frames. Given a fixed path a frame has to follow on a fixed topology, the first parcel of the sum is fixed, independent of any configuration, but not the second one. Therefore, the minimal latency condition is reached when the second part of such sum is zero: when a frame does not need to wait in any queue for the transmission of any other frame.

4.2 Model

Due to the characteristics of the switch, if two messages arrive in a switch from different input ports and, later, leave from two different output ports, they do not interfere with each other in that switch. However, if messages leave through the same output port, they can possibly delay one another due to waiting in the sending queue of that switch. Therefore, as an abstraction of the physical setup, the network model does not need to represent the input ports of each switch, since only the output ports will constrict data flows. In other words, in a switched network, contention points are output ports. Hence, the following definition, illustrated in Figure 4.2, where only the output ports are represented as vertices.

Network topology

Definition 4.9. A network topology is herein defined as the association between the output ports of switches and end-systems of a physical network.

The topology of a switched network where contention points are output ports can be represented as a DAG. A vertex represents an output port and the respective link to the next input port. Each vertex can be named after the physical node where the output port is, and the physical node where the input port is, as explained in the following definition.

Node ($\mathcal{N}$)

Definition 4.10. A node $\mathcal{N}$ is a vertex of a network topology, representing an output port of a physical node, the connected input port, and the link connecting these two. When the node represents the output port of E_i connected to the input port of E_j, it can be represented as $\overrightarrow{E_iE_j}$. A node is capable of emitting, receiving, and/or transferring messages to/from other nodes.

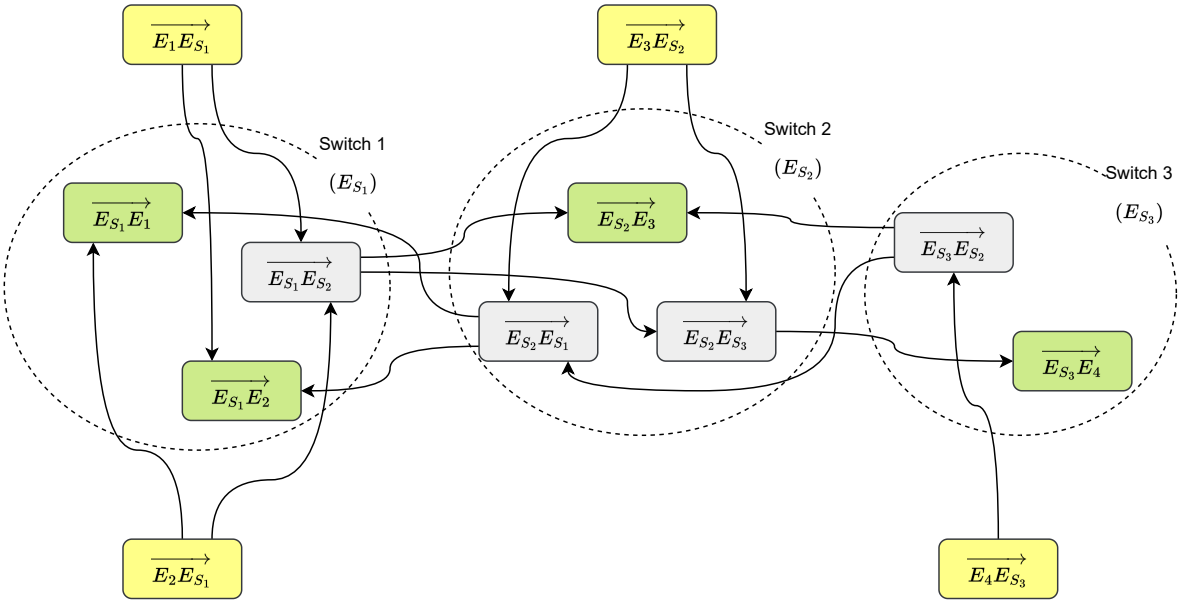

Figure 4.2: Representation of the topology extracted from the example given in Figure 4.1

In Figure 4.2, E_i represents the End-System i, and E_{S_i} represents the Switch i. Nodes producing messages are coloured in yellow, and they always start with an End System in their representation ($\overrightarrow{E_iE_j}$, where E_i is an End System); nodes sending messages to their final input ports are coloured in green, and they always end with an End System in their representation; nodes only transmitting messages from a previous node to a subsequent one are coloured in grey, and only have switches in their representations.

Each end-system has the possibility to send flows of messages to specific target end-systems. Each flow is unique and has a path, and the only configurable parameter is its initial offset. Therefore, we model message flows as follows.

Flow (S_i)

Definition 4.11. A flow S_i is a set of periodic messages (called frames). Its period is T_i and its maximum message size is L_i (in bits), allowing the calculation of its transmission time C_i when the link throughput (in bits per second) is known. It also has a path $\mathcal{P}_i$, a sequence of connected nodes $\{\mathcal{N}\}_i$. An offset O_i might be applied to the flow.

Path ($\mathcal{P}_i$)

Definition 4.12. A path is a sequence of connected nodes $\{\mathcal{N}\}_i$. The nodes are in the form: $\{\overrightarrow{E_{k_0}E_{k_1}}, \overrightarrow{E_{k_1}E_{k_2}}, \ldots, \overrightarrow{E_{k_{n-1}}E_{k_n}}\}$, where E_{k_0} and E_{k_n} are end-systems and the others are switches. Since internal elements appear twice, we can represent the same path as simply $E_{k_0}E_{k_1}\ldots E_{k_n}$.

A flow is therefore very similar to a task, with the additional parameter of a path that it will follow inside the network. Therefore, flows that have paths which do not present any common node will never directly interfere with each other.

Every time a flow passes through a node, it needs to be delayed by the technological delay. Since propagation time and storing time are often much smaller than forwarding time, we simplify and consider that the whole technological delay is composed only of the forwarding time, which is constant with respect to the message size and the topology. Therefore, we work with the following definition.

Store and Forward (S&F)

Definition 4.13. The Store-and-Forward (S&F) delay is a constant technological delay which is at least superior to the maximum storing time of a message in the network.

Also, clock synchronization is considered to be ideal, so there is no significant difference in time measured by any network node. Time can be treated as global.

Then, GCD# can rely on the following considerations:

1. Every flow has a period which is a multiple of a certain common factor Ω. This factor has to be at least equivalent to the maximum transmission time, so GCD# has a chance to reach the minimal latency condition.
2. Only the output ports of switches and end-systems are considered, and will be designated as the nodes of the network. Flows passing through a same node can interfere with each other, while flows that never share a common node cannot directly interfere with each other (even if they pass through the same switch).
3. Paths can merge and fork, but once merged paths fork, they cannot merge again.
4. The time it takes for any frame to be received and resent by any node is constant (S&F);
5. The end-to-end delay is inferior to Ω.

If such considerations are made, the following algorithm is capable of providing a set of offsets to a list of flows in a network, seeking to reach the minimum latency condition.

4.3 Recalling GCD+

In chapter 3, we presented GCD+, a heuristic method that divides the whole execution of a task system into cycles of equal duration. For example, the system whose execution is represented in Figure 4.3 (cycles highlighted with dashed lines) can be seen from the cyclic perspective as it is shown in Figure 4.4.

GCD+ identifies the GCD of all the periods, which is the length of the cycle (represented as Ω), and then reserves specific portions of the cycle for specific groups of tasks (one portion per group, one group per portion), depending on the tasks' subperiods T_S. Such task groups are

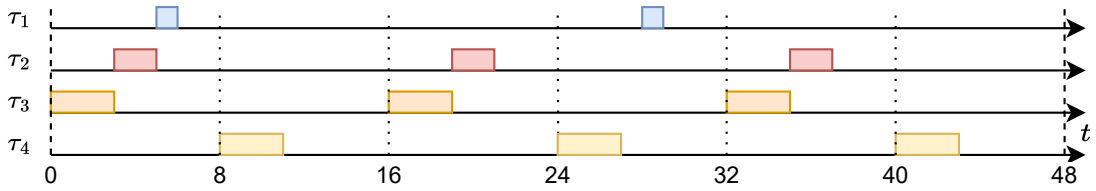

Figure 4.3: Recall of the representation of the execution of strictly periodic tasks in a single processor

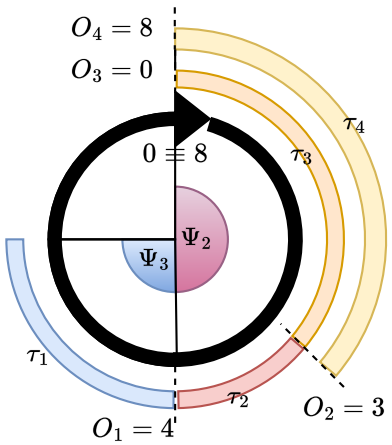

Figure 4.4: Recall of the cycle representation of the tasks of Figure 4.3

called sections and are represented as Ψ. The subperiod is the period T of the task divided by Ω, and if two tasks have co-prime subperiods, they should be put in separate sections. Then, tasks in the same section can alternate cycles, guaranteeing they do not interfere with each other (given certain conditions, as exposed in chapter 3), and tasks in different sections will have different portions of the cycle reserved to them, which also guarantees non-interference given certain conditions.

GCD# is based on the same principles, but the choice of sections and cycles needs to be adapted for the use in networks.

4.4 GCD#

Knowing that an offset can be assigned to each flow in order to better schedule its frames, then GCD# can receive as an input a list of "abstract flows" (defined by their period, message size and path) and return a list of offsets. Based on the same principles as GCD+ (cycles, sections and internal offsets), GCD# operates in five steps:

1. Choosing the section for each flow, based on the value of its period and aiming to reduce

section sizes;
2. In each section, choose a cycle for each flow, trying to reduce the workload in each cycle;
3. Choosing an internal offset for every flow that shares a section, a cycle and a path node with another;
4. Calculating section sizes;
5. Calculating the final offset.

In order to aid the understanding of the algorithm, an example, represented in Figure 4.5, is given as follows (where E stands for End-System, and E_S stands for Switch):

1. S_1: $T_1 = 40$, $C_1 = 3$, Path: E_1-E_{S_1}-E_{S_2}-E_{S_3}-E_4
2. S_2: $T_2 = 30$, $C_2 = 2$, Path: E_2-E_{S_1}-E_{S_2}-E_{S_3}-E_4
3. S_3: $T_3 = 20$, $C_3 = 2$, Path: E_3-E_{S_2}-E_{S_3}-E_4
4. S_4: $T_4 = 60$, $C_4 = 4$, Path: E_3-E_{S_2}-E_{S_1}-E_2
5. S_5: $T_5 = 90$, $C_5 = 2$, Path: E_4-E_{S_3}-E_{S_2}-E_{S_1}-E_2
6. S_6: $T_6 = 20$, $C_6 = 1$, Path: E_1-E_{S_1}-E_{S_2}-E_{S_3}-E_4

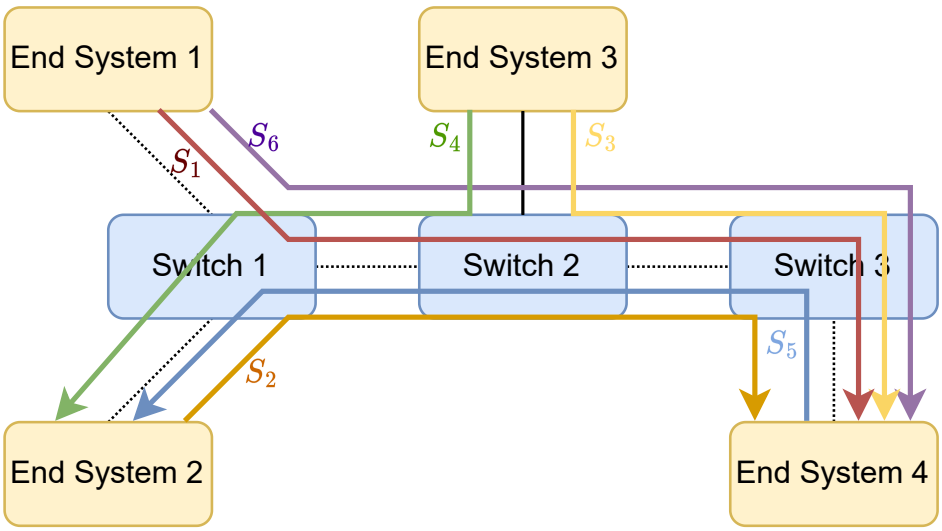

Figure 4.5: Example of flows

In this example, we can quickly see that $\Omega = 10$. Therefore, the respective subperiods are:

1. $T_{S_1} = 4$
2. $T_{S_2} = 3$
3. $T_{S_3} = 2$
4. $T_{S_4} = 6$
5. $T_{S_5} = 9$
6. $T_{S_6} = 2$

Also, the paths to be considered by the algorithm should be composed of the output ports of the system. An output port will be here identified with a composite name: the first half

identifying the physical element to which it belongs, and the second half identifying to where it connects. For example, in the link between End System 1 (E_1) and Switch 1 (E_{S_1}), the output port that belongs to E_1 is identified as $\overrightarrow{E_1E_{s_1}}$, while the output port that belongs to S1 is identified as $\overrightarrow{E_{s_1}E_1}$. Therefore, the output ports for each flow are the following:

1. $\mathcal{P}_1 = \{\overrightarrow{E_1E_{S_1}}, \overrightarrow{E_{S_1}E_{S_2}}, \overrightarrow{E_{S_2}E_{S_3}}, \overrightarrow{E_{S_3}E_4}\}$
2. $\mathcal{P}_2 = \{\overrightarrow{E_2E_{S_1}}, \overrightarrow{E_{S_1}E_{S_2}}, \overrightarrow{E_{S_2}E_{S_3}}, \overrightarrow{E_{S_3}E_4}\}$
3. $\mathcal{P}_3 = \{\overrightarrow{E_3E_{S_2}}, \overrightarrow{E_{S_2}E_{S_3}}, \overrightarrow{E_{S_3}E_4}\}$
4. $\mathcal{P}_4 = \{\overrightarrow{E_3E_{S_2}}, \overrightarrow{E_{S_2}E_{S_1}}, \overrightarrow{E_{S_1}E_2}\}$
5. $\mathcal{P}_5 = \{\overrightarrow{E_4E_{S_3}}, \overrightarrow{E_{S_3}E_{S_2}}, \overrightarrow{E_{S_2}E_{S_1}}, \overrightarrow{E_{S_1}E_2}\}$
6. $\mathcal{P}_6 = \{\overrightarrow{E_1E_{S_1}}, \overrightarrow{E_{S_1}E_{S_2}}, \overrightarrow{E_{S_2}E_{S_3}}, \overrightarrow{E_{S_3}E_4}\}$

4.4.1 Choosing the section

As explained in the previous chapter, the problem of choosing sections is itself a bin-packing problem. We need to choose in which "bin" to put each flow such that the sum of total "bin" sizes is minimized. Some flows will have only one option of section: those whose subperiods T_S are either 1 or a number composed of a single prime number (i.e., for $T_S = p^n$ where p is prime and n is any non-negative integer). If more than one option is available, some considerations must be made, and the flow with the largest message (and, therefore, the largest transmission time) is the first to be assessed in our Best Fit Decreasing algorithm.

If no section amongst the possible choices has any flow assigned to it, the preference is given to the lowest prime. GCD# makes this choice because, the lower the prime, the higher the probability it divides a random integer N. We expect this method to provide a better chance of sharing the section with other flows, avoiding the creation of unnecessary sections and, thus, reducing total section sizes (lemma 3.2.2).

On the other hand, if there is any possible section already occupied by any flow, the preference is given to the occupied sections. This might reduce the need to increase section sizes: adding the flow to a new section would definitely increase the section's size from 0 to C_i, while only in the worst case the already occupied section would need to have it's size increased by C_i. In better cases, the possibility to assign different cycles would allow for a lower (or even null) increase in section size.

Amongst occupied sections, the preference is given according to a score function. The score is a pessimistic estimation of how many occupied cycles there are in the section (since no cycles were assigned yet), hence the section with the lowest score must be prioritized. The estimation has to be made with respect to a candidate S_i, and is given by Equation 4.1. This equation is based on the principle that each flow $S_j \in \Psi_p$ will have a probability of $1/\gcd\{T_{S_i}, T_{S_j}\}$ of occupying a cycle congruent to the cycle assigned to S_i. However, if a score is above the value of 1, we consider that it looses meaning, since there exists a chance that every cycle is already occupied, and the best choices for assigning S_i in this case would

require information we do not have yet. Therefore, the score is limited to 1.

$$\Psi_p.\text{score}(S_i) = \min\left(1, \sum_{\forall j | S_j \in \Psi_p} \frac{1}{\gcd\{T_{S_i}, T_{S_j}\}}\right) \tag{4.1}$$

In our example, each flow has a single option for a section, except for $S4$ ($T_{S_4} = 6$), which can be assigned to either section Ψ_2 or Ψ_3. If the choice for S_4 is made before the choice for other flows, section Ψ_2 is chosen since it represents the smallest prime number 2. However, if both sections are already occupied, the score function must be used. In that case, as seen below, the preference would be given to Ψ_3, since it might be impossible to find a free cycle for S_4 in Ψ_2 (score equals 1), but it is surely possible to find a free cycle for S_4 in Ψ_3 (score of less than 1). Let us consider that S_4 is assigned to Ψ_3, then.

$$\begin{aligned}
\Psi_2.\text{score}(S_4) &= \min\left(1, \sum_{j=1,3,6} \frac{1}{\gcd\{T_{S_4}, T_{S_j}\}}\right) \\
&= \min\left(1, \frac{1}{\gcd\{6,4\}} + \frac{1}{\gcd\{6,2\}} + \frac{1}{\gcd\{6,2\}}\right) \\
&= \min\left(1, \frac{1}{2} + \frac{1}{2} + \frac{1}{2}\right) \\
&= 1
\end{aligned}$$

$$\begin{aligned}
\Psi_3.\text{score}(S_4) &= \min\left(1, \sum_{j=2,5} \frac{1}{\gcd\{T_{S_4}, T_{S_j}\}}\right) \\
&= \min\left(1, \frac{1}{\gcd\{6,3\}} + \frac{1}{\gcd\{6,9\}}\right) \\
&= \min\left(1, \frac{1}{3} + \frac{1}{3}\right) \\
&= \frac{2}{3}
\end{aligned}$$

The non-empty sections are, therefore:

Ψ_2**:** S_1, S_3 and S_6.
Ψ_3**:** S_2, S_4 and S_5.

4.4.2 Choosing a cycle

Once every flow is assigned to a section, the choice for the cycles (O_{C_i}) can be made. Since we consider that the end-to-end delay is inferior to Ω, then each flow S_i will only occupy cycles which are congruent to a unique value $k \bmod T_{S_i}$. Then, for each section, its flows are assigned a cycle according to a decreasing order of their message sizes, as large messages have greater impact.

For each flow, a vector of possibilities is constructed, with size T_{S_i}, populated with zeros. Then, by analysing every flow S_j in the same section which has already been assigned, and checking that they do effectively cross the analysed flow S_i (i.e., their paths have at least a common node), C_j is added to the value in every position in the vector of possibilities that is congruent to $O_{C_j} \mod \gcd(T_{S_i}, T_{S_j})$. Assigning S_i to any of these positions guarantees that S_i and S_j will eventually share the same cycle (lemma 3.2.1). Therefore, the best choice is the position in the vector with the lowest resulting sum – ideally, zero.

In our example, the assignment order is: S_4, S_1, S_2, S_3, S_5, S_6. For S_4, because it is the first flow to be considered, $O_{C_4} = 0$. Then, for S_1, no other flow in its section (Ψ_2) has been assigned to a cycle, so $O_{C_1} = 0$.

However, S_2 belongs to the same section as S_4, and a vector of size $T_{S_2} = 3$ has to be made. In this vector, every cell with a position i congruent to $O_{C_4} \mod \gcd(T_{S_2}, T_{S_4})$ is filled with the value of C_4 (therefore, only the first cell, where $i = 0$, is filled with the value 4). The second cell ($i = 1$) keeps its original value of 0, and is chosen as the assigned cycle. Hence, $O_{C_2} = 1$, the index of the cell with the lowest value.

A similar process occurs for S_3 (resulting in $O_{C_3} = 1$) and S_5 (resulting in $O_{C_5} = 2$). For S_6, however, the option where $O_{C_6} = 0$ is already occupied by S_1, and the only alternative ($O_{C_6} = 1$) is also occupied by S_3. In that case, the internal offset also has to be considered before making a choice.

4.4.3 Choosing an internal offset

In parallel to each cycle choice, an internal offset is assigned to each flow S_i. For that, we only need to consider flows S_j that fit all the following constraints:

1. S_j crosses S_i (they have a common output port);
2. S_j and S_i are in the same section;
3. S_j and S_i are assigned to congruent cycles (modulo $\gcd\{T_{S_i}, T_{S_j}\}$);
4. S_j has already been assigned an internal offset.

If no such S_j exists, then $O_{I_i} = 0$, which is the case when an empty cycle is assigned to a flow.

However, the S&F constant needs to be considered. If the path of S_j shares a node $\mathcal{N}$

with the path of S_i, and $\mathcal{N}$ is the n^{th} node for both the respective paths, then both flows will experience the same S&F delay. In this case, the flows must be analysed as if there was no S&F, and the internal offset is chosen such that there is no overlap between the messages. Mathematically, this means that an O_{I_i} shall be chosen such that the following conditions are true:

$$\begin{aligned} O_{I_i} \leq O_{I_j} &\Rightarrow O_{I_i} + C_i \leq O_{I_j} \\ O_{I_i} > O_{I_j} &\Rightarrow O_{I_i} \geq O_{I_j} + C_j \end{aligned} \tag{4.2}$$

However, if the common node $\mathcal{N}$ is the n^{th} node for a path of S_i but the $(n+k)^{\text{th}}$ for S_j, this means that, when their frames arrive in $\mathcal{N}$, S_j will be late by k times S&F with respect to S_i. Therefore, for every flow S_j that comply to the aforementioned conditions, there needs to be a correction. If S_j is in advance with respect to S_i, we must subtract the proper amount of S&F delays from O_{I_j} in the conditions in Equation 4.2. If it is late, we must add the amount of S&F delays to O_{I_j}.

In the case of S_6 for example, the algorithm can evaluate each possible choice and decide to assign the flow to the cycle where the internal offset O_{I_6} is the smallest. For the first option $O_{C_6} = 0$, there is no difference between the amount of hops S_1 and S_6 have to make to arrive to the same node, so Equation 4.2 is valid without corrections. Since $O_{I_1} = 0$, then the smallest value for O_{I_6} that satisfies Equation 4.2 is equivalent to C_1, i.e., 3. For the second option ($O_{C_6} = 1$), the node S2S3 is the second node in $\mathcal{P}_3$, but the third in $\mathcal{P}_6$. This means that, if sent at the same time, a frame from S_3 arrives earlier than a frame from S_6 by the equivalent of one S&F constant. If we consider that, in our example, S&F is equivalent to the maximum transmission time (i.e., 4), then O_{I_6} can equal zero (which is preferable). This is shown in Figure 4.6.

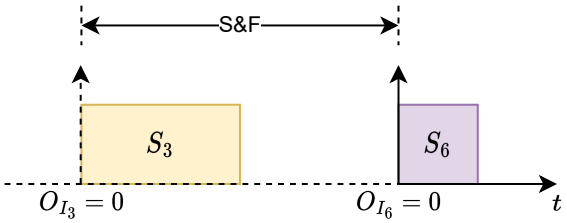

Figure 4.6: Example showing the difference in arrival time on node S2S3 due to the constant Store and Forward time

As it can be seen in the example, considering the S&F time delay translates to shifting the positions of other flows along the time axis. Then, once again, a free slot must be sought, and the internal offset O_{I_i} can be defined for the flow S_i.

4.4.4 Calculating section sizes and the final offset

In order to calculate each section size $\Psi_i.\delta$, the execution order between sections must be defined. GCD# arbitrarily chooses the order based on the number p related to the section: if $p_1 < p_2$, then messages of flows belonging to Ψ_{p_1} are sent before those that belong to Ψ_{p_2}.

The size of any section Ψ_{p_i} can be calculated in a number of steps, using the following section Ψ_j.

1. A table is constructed such that each pair $(S_i, S_j) \in \Psi_i \times \Psi_j$ is represented;
2. The S&F difference for each pair is calculated (considered to be zero if $\mathcal{P}_i$ and $\mathcal{P}_j$ do not cross), and added (if S_i is late) or subtracted (if S_i is early) from the value of $O_{I_i} + C_i$;
3. The maximum resulting value among all pairs is the section size of Ψ_i.

For the last section, the first section is considered to be the "following" section.

In our example, to determine the size of Ψ_2, we need to evaluate all the pairs resulting from the cartesian product $\{S_1, S_3, S_6\} \times \{S_2, S_4, S_5\}$. From those nine elements, only (S_3, S_2) has a non-null S&F value. Therefore, the three pairs whose first element is S_1 will result in a value of $O_{I_1} + C_1 = 3$. Similarly, for the three pairs whose first element is S_6, we obtain the value of 1. For (S_3, S_4) and (S_3, S_5), we obtain, in the same manner, the value of $O_{I_3} + C_3 = 2$. However, for (S_3, S_2), we have to subtract the S&F constant from the result, since S_3 is early with respect to S_2. The corresponding value for the analysed pair is, therefore, -2. Finally, the section size is obtained as the maximum calculated value: $\Psi_2.\delta = 3$.

Now, to determine $\Psi_3.\delta$, we do the same process, but considering Ψ_2 as the section following Ψ_3. Therefore, the pairs result from the following set multiplication: $\{S_2, S_4, S_5\} \times \{S_1, S_3, S_6\}$. From those nine elements, only (S_2, S_3) has a non-null S&F value. With that exception, every pair whose first element is S_2 has a resulting value of 2; every pair whose first element is S_4 has a resulting value of 4; every pair whose first element is S_5 has a resulting value of 2. Specially for (S_2, S_3), we need to add 4 to $O_{I_2} + C_2$ since S_2 is late with respect to S_3, resulting in the value of 6. Therefore, $\Psi_3.\delta = 6$, the maximum calculated value.

If section sizes add to a value which is not greater than Ω, then GCD# has found a null-interference configuration for the system: no frame will have to wait for another frame to be sent (which is the case for our example). Otherwise, GCD# is still able to provide a proposition of offsets, even though there will probably be an additional waiting time for some frames.

After section sizes are calculated (the last section being considered to precede the first section due to the cyclicity), then every O_S can be calculated. For every section, O_S is the sum of the sizes of preceding sections (disregarding the cyclicity). In other words, the O_S for the first section is 0; for the second, it equals the size of the first section; for the third, it equals the sum of the sizes of the two first sections; and so on.

Therefore, the offsets for every flow can finally be calculated according to Equation 3.8. Based on them, analyses and simulations can be run on every node in order to generate the

schedules. In our example, the offsets are:

- $O_1 = 10 \cdot 0 + 0 + 0 = 0$
- $O_2 = 10 \cdot 1 + 3 + 0 = 13$
- $O_3 = 10 \cdot 1 + 0 + 0 = 10$
- $O_4 = 10 \cdot 0 + 3 + 0 = 3$
- $O_5 = 10 \cdot 2 + 3 + 0 = 23$
- $O_6 = 10 \cdot 1 + 0 + 0 = 10$

4.5 Case Study

The case study was chosen to be Orion, for which the required information was obtained from [Zha+20]. Its physical topology is represented in Figure 4.7.

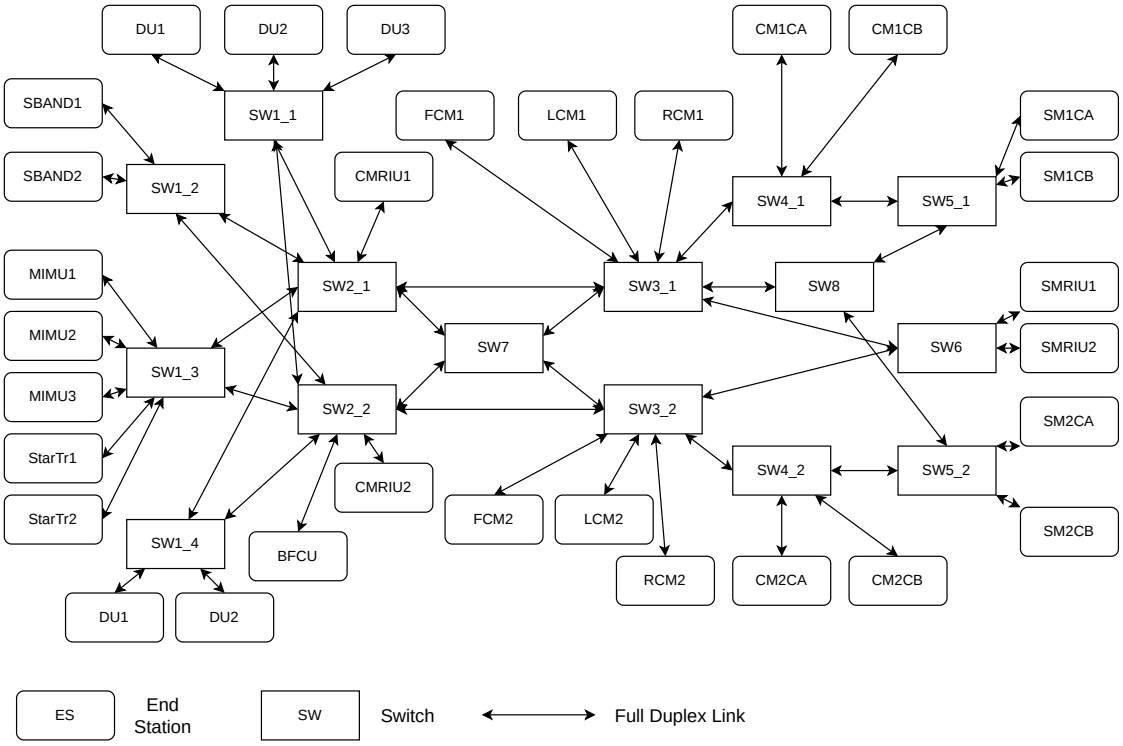

Figure 4.7: Physical topology of Orion

The flows used were those in the Realistic Test Case, containing 100 Time-Triggered flows. The network is considered to be operating at 1 Gb/s, meaning that the time-unit of the system is 1 ns (transmission time of one bit). The S&F time is considered to be constant, in this case equal to the maximum transmission time of a frame of the network. In the case of Orion, it is equal to 12216 nanoseconds (equivalent to the transmission time of 1527 bytes, the largest frame of the model).

The algorithm of GCD# was implemented using Python 3.9 and run on a laptop with

Ubuntu 18.04.1, Intel Core i7-4710MQ CPU (2.50GHz, 4 cores, 8 threads), and 16 GB of RAM.

Using this implementation, offsets were calculated in under 40 ms for all the flows. The value for Ω is found to be 625000 ns, and the total section size is 109800 ns, inferior to Ω. This is an indicator that, in the considered model with constant technological delay for each switch, each frame is sent as soon as it is ready, without the need to wait for any other frame in the queue. If section sizes summed up to a value larger than Ω, there would probably be an interference between frames, resulting in frames waiting in the queue for other frames to be sent.

Later, for verification purposes, the solution is evaluated according to a pairwise analysis: each possible pair of flows is analysed and, if they cross in any node, their estimated arrival times at that node (offsets + S&F) are calculated in order to verify if there exists any overlap, i.e., if any message should arrive at the node before the other one is completely sent. After an analysis made in each node for each pair of tasks, no overlap was identified, i.e., there is no interference whatsoever. Therefore, in the modelled network (constant S&F), no message would need to wait in the output queue for another message to be sent, and the minimum latency condition is met.

On the other hand, if we are in a different configuration where the total section size would be superior to Ω, interferences might happen. Further analyses, using other techniques, would be required to evaluate the behaviour of the network.

4.6 Conclusion

We proposed a heuristic method called GCD#, which proposes a set of initial offsets for a set of strictly periodic frame flows. GCD# seeks to (but does not require to) completely avoid interference in a switched network. It is based on GCD+, itself based on decomposing a larger NP problem into smaller NP sub-problems. For such, GCD# (as well as GCD+) divides the time in cycles, sections and offsets within sections in order to exploit the semi-harmonicity of the periods.

Using the offsets given by the heuristic and the algorithms to find the cycle, the user of GCD# can potentially create a correct schedule for a time-critical network without needing to rely on time-consuming and error-prone techniques such as optimization tools. Comparing with methods using constraint programming such as [RPC20], that do not directly seek to minimize latency, may take several seconds or even several minutes for large samples, and may not find a feasible solution for the problem, GCD# presents itself as a scalable option that can always propose a solution, even if it does not guarantee the minimum latency condition.

In the future, it is possible to include in GCD# a more realistic approach for the technological delay, with a value that depends on the size of the message. Also, improvements can be made in order to account for limits in the number of queues available per switch.

Networks in which frame flows have asynchronous arrival patterns can also be a target for future improvements. In addition, better tooling (proper simulations and network calculus, often available under closed software licences) could be used to perfect the evaluation of each offset generation method.

Part II

Model-Driven Engineering for Drones

Chapter 5

Background: Model-Driven Engineering and Software Architectures

Drones have been under intense development for several years, and their systems have been continuously increasing in complexity. Their autopilot code, for example, might be composed of millions of lines.

Assessing the safety of such complex systems is not trivial. Even with the help of current techniques to evaluate certain architectures, simply extracting the autopilot's architecture might prove to be a very hard and error-prone task. Therefore, techniques to abstract software systems are key to guaranteeing the nominal functioning of drones and the accomplishment of their intended tasks.

In this chapter, we provide a background in one of the most convenient family of methods responsible for abstracting software systems: the Model-Driven Engineering.

5.1 Model-Driven Engineering

Model-Driven Engineering (MDE) (in a broad sense equivalent to Model-Based System Engineering (MBSE)), according to [MCF03], "is simply the notion that we can construct a model of a system that we can transform into the real thing." A model can be defined as a coherent set of abstract formal elements that describe a real system according to a certain perspective. The perspective will define which characteristics of the system will be described, and which ones will be ignored. An implementation of an algorithm in C, for example, can be seen as a model for the compiled code, as well as a flowchart that represents this algorithm.

Every model needs to comply with a set of formal rules so that it can carry a consistent meaning. This set of rules can be expressed as a metamodel: a more abstract model that describes the meaning of a set of models (the metamodel's instances). A defined metamodel allows its instances to be in accordance with the same set of concepts, notations, processes and tools [BCW17]. A sufficiently complete metamodel might allow the transformation of its instances into instances of another metamodel automatically. The transformation can be itself represented as a model.

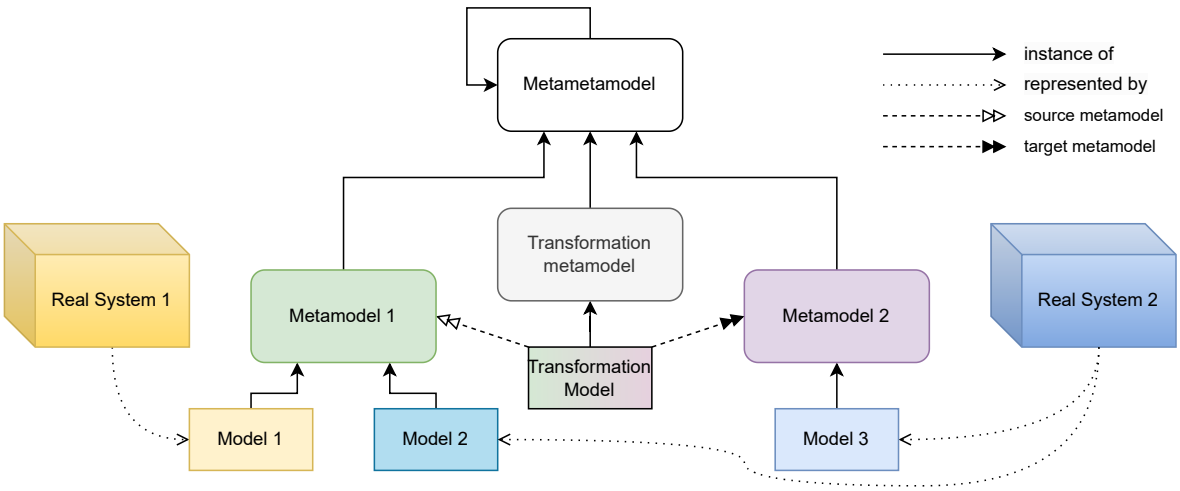

Figure 5.1: Models, metamodels, transformation models and their relations

The aforementioned concepts are represented in Figure 5.1, adapted from [Jou+08]. A system (the left cube, in orange) is represented by a model (Model 1), which conforms to a certain metamodel (Metamodel 1). The other system (the right cube, in blue) is represented by a model (Model 2), which conforms to the same metamodel as Model 1. However, a second metamodel is proposed, and a transformation model exists between the two metamodels. This means that, using the rules defined by the transformation model, Model 3 can be created from Model 2. This new model conforms to a different metamodel, but also represents the system in blue. Also, both metamodels conform to the same metametamodel, which conforms to itself in this example.

Being models an abstraction of real systems, and metamodels an abstraction of models themselves, metamodels can be used as very powerful tools. They define the structure in which models can be instantiated, i.e., their concepts and relations. These take form in a specific grammar that dictates how a complying model must be made. Hence, metamodels contain the perspective from which real systems should be viewed in order to be modelled. When a metamodel is used as such, it can be called a Design Specific Modelling Language.

5.2 Domain Specific (Modelling) Languages

In order to represent systems in the same knowledge domain and regarding the same perspective, a metamodel can be created and used as the preferred reference to create system models. Since every model will conform to the formal rules expressed by this metamodel, one could argue that the metamodel is itself a type of grammar, capable of describing unambiguously a set of systems.

Therefore, a programming language can be created to express the elements and relations of the metamodel's instances. The language, called Domain Specific Language (DSL), can then

be parsed by suitable transformation models in order to generate other models or even the objects it represents. Examples of DSLs are HTML, LaTeX and SQL. In contrast to General Purpose Languages (such as C++ or Java), DSLs only have meaning when used to represent the application domain they were designed for.

Since DSLs embed the concepts and relations of the represented systems in the languages themselves, they allow their users to remain in a higher abstraction level when in the conception phase. In other words, the users of DSLs are spared of (most) implementation details required when a system needs to be implemented in any General Purpose Language. Using an algorithm as an example of a modelled object, details such as memory allocation and variable specification can be ignored by the user, since the transformation model will be responsible for translating the higher-level concepts into lower-level elements. Thus, DSLs have the potential to significantly ease the design and evaluation of the objects they represent.

In order to ease even further the use of such languages, a possible approach is to use Domain Specific Modelling Languages (DSMLs). These are DSLs used specifically to create models of the represented object. In order to do so, a graphical interface can be implemented to aid the user of the DSML. Using graphical interfaces is often more intuitive and, therefore, requires a less-challenging learning process compared to using textual languages.

On the other hand, using DSLs has also its drawbacks, as well put in [Sel07]. The more specific a language is, the less support infrastructure it has (compilers, editors, debuggers, etc.), the fewer experts there are in that language (in opposition to a great availability of experts in the most known languages) and the fewer developers there are to expand and adapt the language. Yet, some DSLs can present more advantages than disadvantages in certain situations, and have become a reference for specific domains, such as the following examples.

5.2.1 AADL

Proposed by the Society of Automotive Engineers (SAE) in 2004 as the standard AS5506 [Fei+04], the Architecture Analysis and Design Language (AADL) is a DSL conceived to represent the elements of an avionics system. Nowadays, it consists of a graphical and textual language able to describe real-time embedded systems from both physical (hardware) and logical (software) perspectives. Each view represents the components that constitute the system, and components can connect to each other via ports and other connections.

In the hardware view, the following components can be instantiated (their visual representations shown in Figure 5.2):

- system
- processor
- device
- memory
- bus

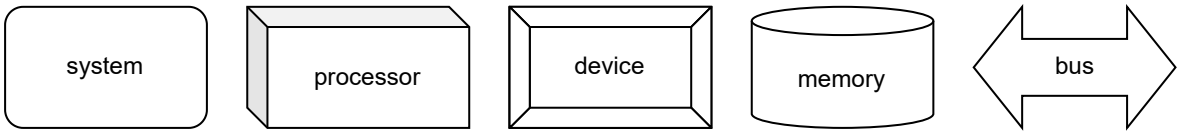

Figure 5.2: AADL representations of hardware components

In the software view, the possible components (and their visual representations shown in Figure 5.3) are:

- system
- data
- subprogram
- thread
- thread group
- process

Figure 5.3: AADL representations of software components

Each one of the aforementioned components can nest a specific set of elements. Also, for interfacing other components, three kinds of ports (visual representations shown in Figure 5.4) are allowed:

- event port
- data port
- event data port

Figure 5.4: AADL port representations

One of the tools that support AADL is Open Source AADL Tool Environment (OSATE) [Fei19], an Eclipse plugin that allows the user to instantiate AADL entities with a graphical interface or textual descriptions.

An alternative to AADL is the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) language [Fau+07], a Unified Modeling Language (UML) profile that supports specification, design and validation of real-time architectures. It has been standardized by the OMG, and it extends UML capabilities in order to represent hardware and software characteristics. Modeling and Analysis of Real-Time and Embedded Systems (MARTE) does not define a specific modelling process, so it is possible to use different elements in different contexts.

5.2.2 MARTE

UML is a modelling language intended to provide a standard way to visualize the design of a digital system. It supports different types of diagrams (e.g., structural, behavioural) that can be interrelated amongst them, and together provide different perspectives of both a global view and specific parts of the system. It is also a general language, providing a graphical syntax for their elements. In addition, it provides an extension mechanism: the profile, which can be interpreted as a way to create a (graphic) DSL on top of UML.

The profile for model-driven development of Real Time and Embedded Systems, i.e., the MARTE profile [Fau+07], is an example. An autopilot, and the navigation components on board the drone in general, are real-time embedded systems, with specific performance constraints, a typical software architecture, a resource constrained hardware platform architecture, and a specific mapping of the software to the hardware architecture.

The MARTE profile provides a number of extensions, structured on a number of packages. The profile has a "foundation" set of packages which enable time related annotations and non-functional properties. The foundation packages also support the generic modelling of components, high-level functionalities and resources, as well as the allocation among them. From this foundation, more specialized extensions are provided.

5.2.3 Drone Mission Tools

Instead of describing the architecture of an embedded system, these tools aim specially at describing the use of functionalities provided by the drone. They are described in the following paragraphs.

RoboChart is a DSML for robotics that represents sets of state machines [Miy+19]. This modeling tool is built on formal semantics that enable the verification of the models. It provides constructs to model real-time concurrent designs and includes the notions of robotic platforms with distributed controllers, which may contain parallel threads.

FlyAQ [FLY19] is a tool able to automatically generate a complete Python code for executing a mission. Using two different sets of objects on a map, a mission specification (trajectories, regions of interest and activities) is merged with a context definition (geofencing and obstacles) to automatically generate a sequence of functions that complete the designated mission taking the constraints into account. Then, this sequence is automatically transformed in a python script that commands a drone fleet over a GCS using DroneKit [Dro15], a Python implementation of MAVLink commands. It is important to note that the generated application is meant to be only executed on a GCS.

FlytOS is an operating system built on ROS and Linux, made to facilitate the development of drones [Fly19]. It provides APIs and SDKs for building high-level applications, and interacts with the drone/autopilot through drone-adaptors, exposing the high-level APIs in ROS, C++, Python, REST and Websocket.

RobMoSys [Rob19a] is a project willing to create a design approach that manages the interfaces between different developer roles, thus separating concerns, in an efficient and systematic way. Its proposal envisages a set of fully model-driven methods and tools for composition-oriented engineering of robotics systems. Following this approach, SmartMDSD [Den+16] is an IDE capable of modelling robotic systems, and automatically generating the skeletons of ROS systems that can be manually completed and compiled.

The ReApp project [Wen+16] is an approach that uses MDE and ROS to generate and to arrange software components so that a robot behaviour can be instantiated.

SmartSoft [Lot+17]; [Bru15] is a framework for using different software components to compose a robotic system, but also providing tools for optimizing the design.

RobotML [Dho+12] was part of the now finished PROTEUS project. It uses robot ontologies to provide "a common ground for designing and implementing component-based robotic systems". This approach focuses on deploying code to OROCOS-RTT5, RTMaps6, Urbi7 or Arrocam (which were defined by the Proteus project).

The BRIDE framework [BRI14] is a modelling tool that generates code from an abstract representation of algorithms and from the model of a ROS system. Some similar solutions intended for robotic behaviour, but possibly useful for drones, include CARMEN [MRT03] and MARIE [Cot+06].

5.3 Technological Background: Software Buses

Different strategies for implementing communication channels between software entities can be seen as software buses. They abstract implementation details and provide the application layer with a powerful Application Programming Interface (API) which allows the modularization of the application code, as well as its independence from the hardware. Although some of these technologies have been described in chapter 2, let us recall them from a modelling perspective.

5.3.1 MAVLink

To allow the autopilot to communicate with external entities, the Micro Air Vehicle Link (MAVLink) communication protocol [Dro19] has become a de-facto standard to send and receive messages between an unmanned vehicle and a remote GCS. The protocol defines a lightweight header-only message set and structure, being open for extensions. This allows a pilot to control the drone during its flight, and also the evaluation in a ground station of some parameters.

A MAVLink header contains:

1. the start-of-frame byte (0xFE);
2. the length (integer between 0 and 255);

3. a sequence indicator, allowing to detect packet losses (integer between 0 and 255);
4. the ID of the sender system (integer between 1 and 255);
5. the ID of the sender component (integer between 0 and 255);
6. the ID of the message (integer between 0 and 255), allowing the interpretation of the payload.

Each autopilot compliant to MAVLink contains a MAVLink serialization module responsible for transmitting data according to the protocol (from the drone to a GCS and vice-versa). For implementing MAVLink communication in a ROS system, a translation module (from ROS topics to MAVLink messages) can be used, for example MAVROS [Erm18].

5.3.2 ROS

Robot Operating System (ROS) [Rob19b] is a middleware conceived to be used on robots, summarizing most of the concepts used in robot behaviour projects and easing the development of multitasking applications with both synchronous and asynchronous communications.

As sketched in Figure 5.5, a system running on ROS has parallel execution nodes that communicate to each other by topics (asynchronous communication) or services (loosely synchronous communication).

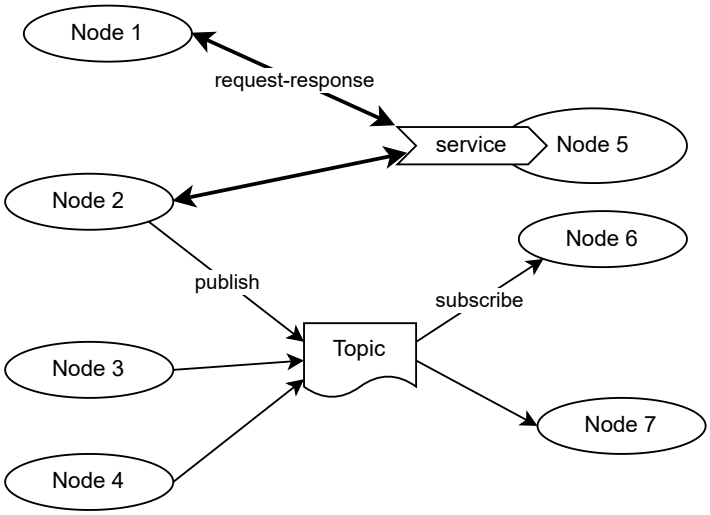

Figure 5.5: Example of a ROS system containing nodes, a topic and a service

In order to publish or subscribe to any topic, or to use any service, a ROS node needs to register its intentions to the ROS Master node, a centralized executable that manages all the communication in the ROS system.

Nodes, topics and services can be assembled in ROS packages, which can be provided as a whole and few steps away from deployment.

5.3.3 ROS2

Robot Operating System 2 (ROS2) [OSRF19] is a different approach used to express the core concepts of ROS. Therefore, it still relies mostly on the concepts of nodes and topics, but as a decentralized architecture, it does not require a ROS Master node. Instead, it relies on its communication layer discovery protocol to establish every necessary connection between publishing and subscribing nodes.

Robot Operating System 2 (ROS2) communication layer is a data-centric publish-subscribe standard published by the OMG, called Data Distribution Service (DDS) [Gro15]. DDS is an abstract architecture, having several implementations, each one distributed by a different provider (e.g., OpenDDS, eProsima's FastDDS and RTi Connext). This technology has been used in several critical applications such as battleships, dams, flight and space systems, thanks to its Quality of Service (QoS) policies, for example regarding deadline, durability, latency, virtual partitions, reliability, and othres.

DDS uses Interface Description Language (IDL) to provide a decentralized publish-subscribe transport (with similar concepts to ROS such as Topics, Publishers and Subscribers) and an extension for decentralized request-response transport. In order to guarantee the interoperability between DDS systems deployed onto different pieces of hardware connected in the same network, DDS uses Data Distribution Service Interoperability Wire Protocol - Real-Time Publish-Subscribe (DDSI-RTPS) [Gro19a] as its serialization protocol. DDSI-RTPS itself relies on top of transport layers such as UDP or TCP. The complete layered structure is represented in Figure 5.6.

5.3.4 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 sensor data from the low level drivers to the state estimation filters, but it is also widely used by payload modules to retrieve or send data and commands.

The main concepts behind ABI are:

- Provide an easy way to allow software components to exchange data with minimum delay and execution overhead;
- Application modules are agnostic with respect to each other, such that the only important information they need is the data format;
- Each subscriber sets a callback function to be called when new data arrives.

All the possible messages are described in an XML file that specifies the name and identifier of the message, as well as the names and types of its fields. A code generator produces a C code from these definitions, which consists in callback prototypes, binding and sending functions.

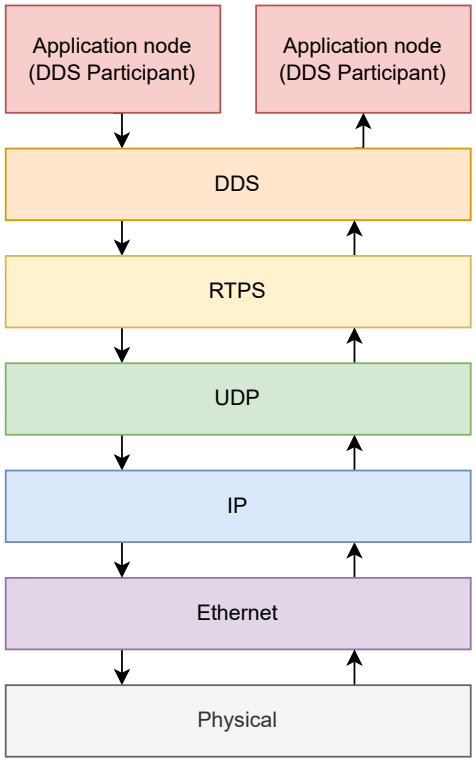

Figure 5.6: DDS Layers

Each sender has a unique identifier. When binding, a subscriber can filter messages by selecting a specific source, accept all 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, ABI is not thread-safe, and it should be used only inside the main autopilot thread, ignoring the underlying RTOS (if any).

5.4 Architecture and Methodology Inspiration

5.4.1 Arcadia

The Architecture Analysis and Design Integrated Approach (Arcadia) [Roq16], 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 Standardization 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 5.7[1]. This methodology is well suited for development approaches based on the V cycle.

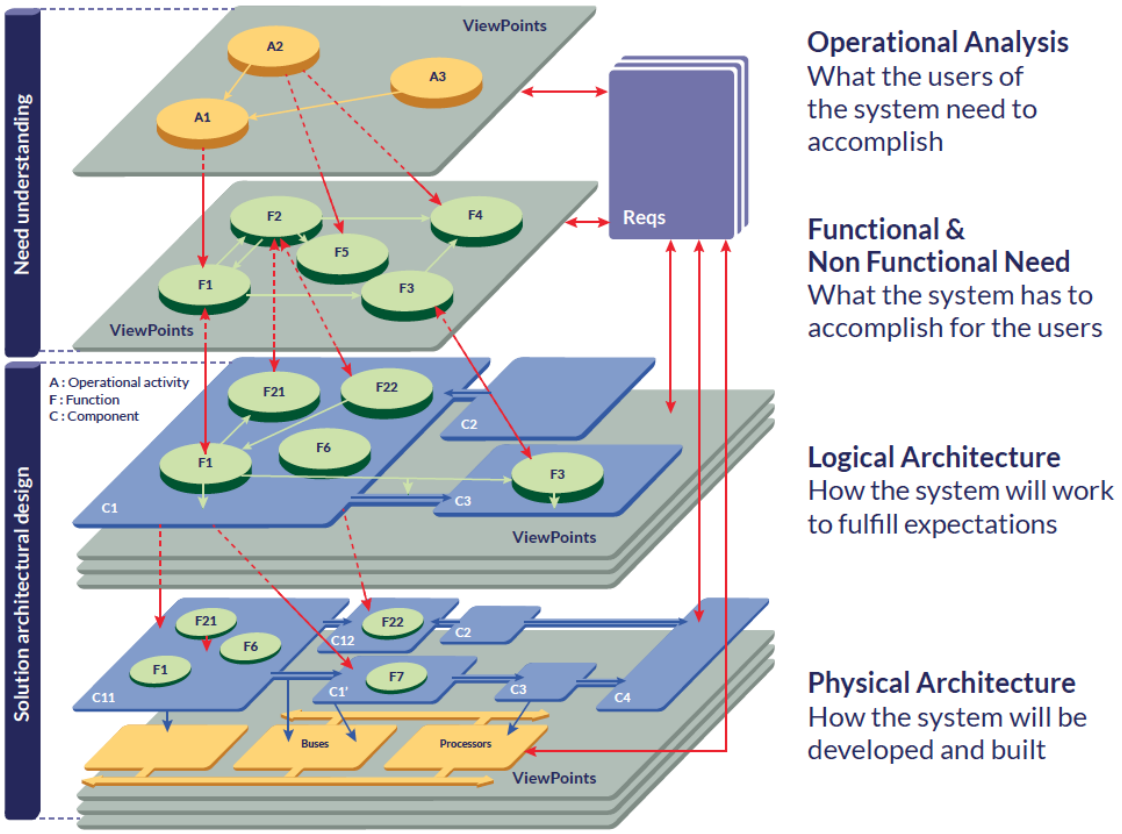

Figure 5.7: Arcadia Method representation

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.

5.4.2 RobMoSys

The Robotics Model-Based System Design (RobMoSys) project [EU17] was funded by the European Commission as part of its Horizon 2020 program. The project involved 13 partners from both industry and academia, including universities, research institutes, and companies such as Siemens and Bosch. Its main goal was to establish a common framework for developing and integrating robotics systems, with a focus on model-based engineering. Its general principles are shown in Figure 5.8[2].

[1] `https://www.eclipse.org/capella/arcadia.html`
[2] `https://robmosys.eu/wiki/`

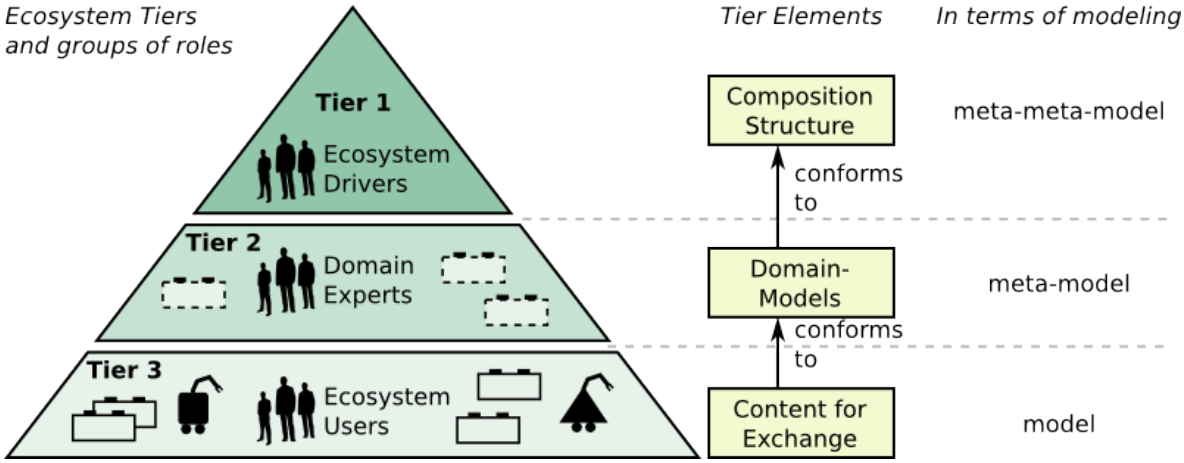

Figure 5.8: RobMoSys general principles

RobMoSys aimed to apply model-based engineering to the development of robotic systems, which are typically complex and composed of many different components, such as sensors, actuators, and controllers. The project focused on creating a standard methodology for designing and composing these components, using a set of common models and interfaces. By standardizing the way in which robotic systems are designed and composed, the project aimed to reduce the complexity and cost of developing and maintaining such systems. It also aimed to create a set of open-source tools and software libraries to support the development of robotic systems. These tools and libraries are freely available to researchers and developers, and are designed to be compatible with the RobMoSys methodology.

Chapter 6

RoBMEX

At first, we can use MDE to provide the means to use a drone autopilot as a black box: a COTS (Component Off-The-Shelf, in contrast with MOTS - Modifiable Off-The-Shelf). Adding functionalities to the drone requires developing an application outside of the autopilot that continuously communicates with it. The validation of the embedded system, in this case, will rely on the thousands of hours of flight that same system will have made among all its users around the globe. With this perspective, the ROS-Based Modelling Framework for End-Users and Experts (RoBMEX) framework [LOG21] was developed.

RoBMEX was constructed around the concept of a drone mission. Its goal is to aid developers to easily configure a specific drone behaviour without the need to change the autopilot.

Two main kinds of users are considered for the RoBMEX framework:

- the end-user, who has a little or no knowledge of programming, but wants to order the vehicle to do something of his/her interest;
- the programmer, who can develop specific functions to be used by the vehicle, but has little experience with integrating drone technologies.

These two users interact around the concept of drone missions: the desired behaviour of the drone. The end-user has an objective in mind, while the programmer will provide the tools and building blocks which will be used to compose certain behaviours. Therefore, we shall construct the framework around this same concept.

Mission

Definition 6.1. A mission is a set of actions that can be performed by an agent (in our case, a drone). These actions are interconnected by conditions, which determine when one action finishes and other ones start. The mission thus holds an ordering to its constituent parts.

Given the nature of most programmed behaviours, we consider that the set of actions that compose a mission can be arranged in parallel, in a sequence, or in more complex subsets combining these two fundamental arrangements. Based on this topology, a set of metamodels was designed. The design is explained in the following sections.

6.1 RoBMEX methodology

At first, we consider that there exists a ROS system able to link a drone to a ground station in a computer. RoBMEX must be able to create launchable node codes to work in this environment, translating a specified mission using a graphical language. The RoBMEX instance will then communicate with the autopilot software through MAVLink messages, as depicted in Figure 6.1. Note that RoBMEX instances can also be deployed on companion boards if they communicate with the autopilot using MAVLink.

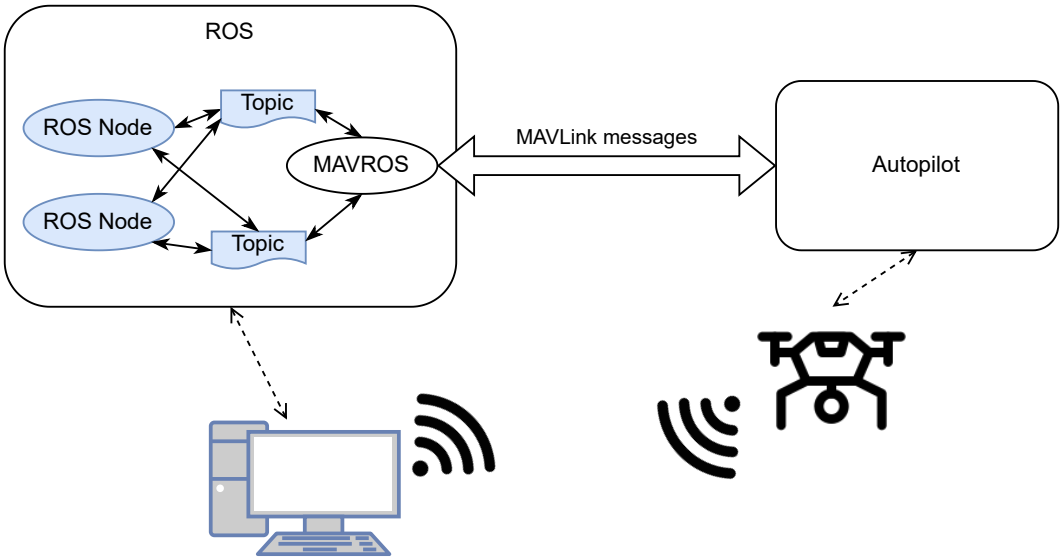

Figure 6.1: Representation of RoBMEX instances in blue on a remote GCS

Based on the underlying ROS technology, a metamodel expressing its concepts is designed, called ROSModL. This metamodel uses the middleware's basic structures, and is independent of any other metamodel created.

For the programmer, it is interesting to be able to create simple processes, such as variable comparisons or other operations, and to save them in a library. Another user, like an end-user with no expertise, would access this library to compose their mission. For this, common operations such as mathematical calculations, variable definitions, logical evaluations and block structures (conditionals and loops) have to be modelled. This metamodel, called ROSProML in the RoBMEX framework, has to use some concepts of the ROS middleware, which means that it needs to be dependent on ROSModL.

At last, composing a mission needs not only the processes built on ROS , but connections defined between their inputs and outputs. The end-user must be able to organize the whole execution of the mission, without deep knowledge regarding the system details. That is why a metamodel of a mission, based on the previous two metamodels, has to be created. It is called ROSMiLan, and it expresses the given definition of a mission.

The three DSMLs interact in dynamic layers, as illustrated in Figure 6.2, composing the

RoBMEX framework. ROSProML, represented on top, defines the behaviour of an execution node, the operations made with the input variables, the functions being called, and the values of the outputs. ROSModL, on the bottom, is responsible for modelling ROS systems, with its nodes and topics. ROSMiLan, between the other two, connects mission elements. For instance, a task (in a multitask mission expressed in ROSMiLan) is related to a single ROS node, and is connected to inputs and outputs that can be represented as topics in ROSModL.

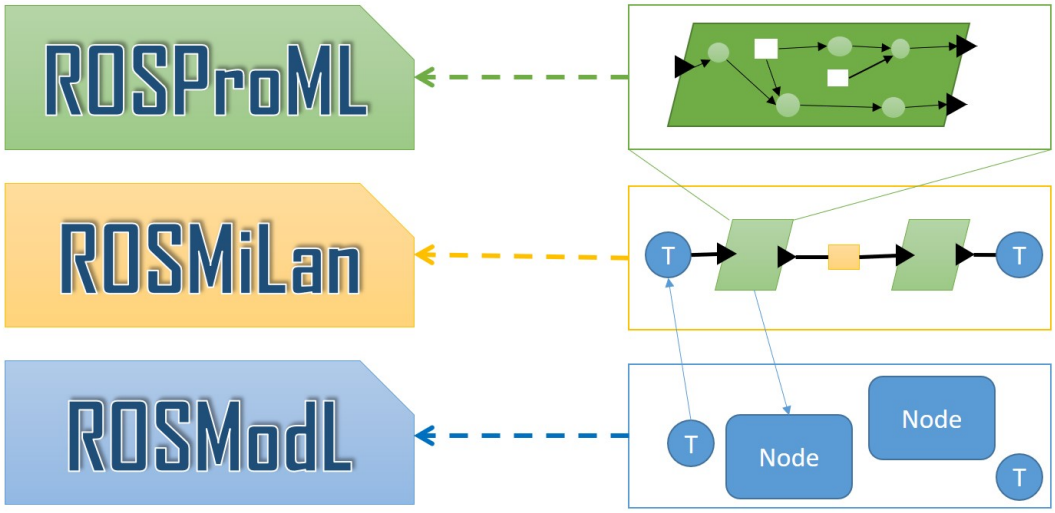

Figure 6.2: The three RoBMEX DSMLs

6.2 RoBMEX foundations

Every DSML is based on two main concepts: an abstract syntax (the metamodel structure) and at least one concrete syntax, which can be textual, graphic or both of them. Therefore, this section is dedicated to detail the content of each one of the three RoBMEX DSMLs' abstract syntax. The following metamodels are expressed in Ecore language[1], which belongs to the Eclipse IDE (Integrated Development Environment).

6.2.1 ROS Modelling Language (ROSModL)

ROSModL is an abstraction of any ROS system. It focuses on ROS's key-features such as nodes, topics and services. Its metamodel is illustrated in Figure 6.3.

`RosSystem` is the root class of ROSModL. Every instance of `RosSystem` has an attribute `version` indicating the ROS version the designer wants to use (e.g., Indigo, Jade or Kinetic). This is very helpful to generate adequate code. Also, every instance of `RosSystem` is composed of a set of `RosPackage` instances. Each instance has a defining name, a path to its location

[1] `https://www.eclipse.org/modeling/emf/`

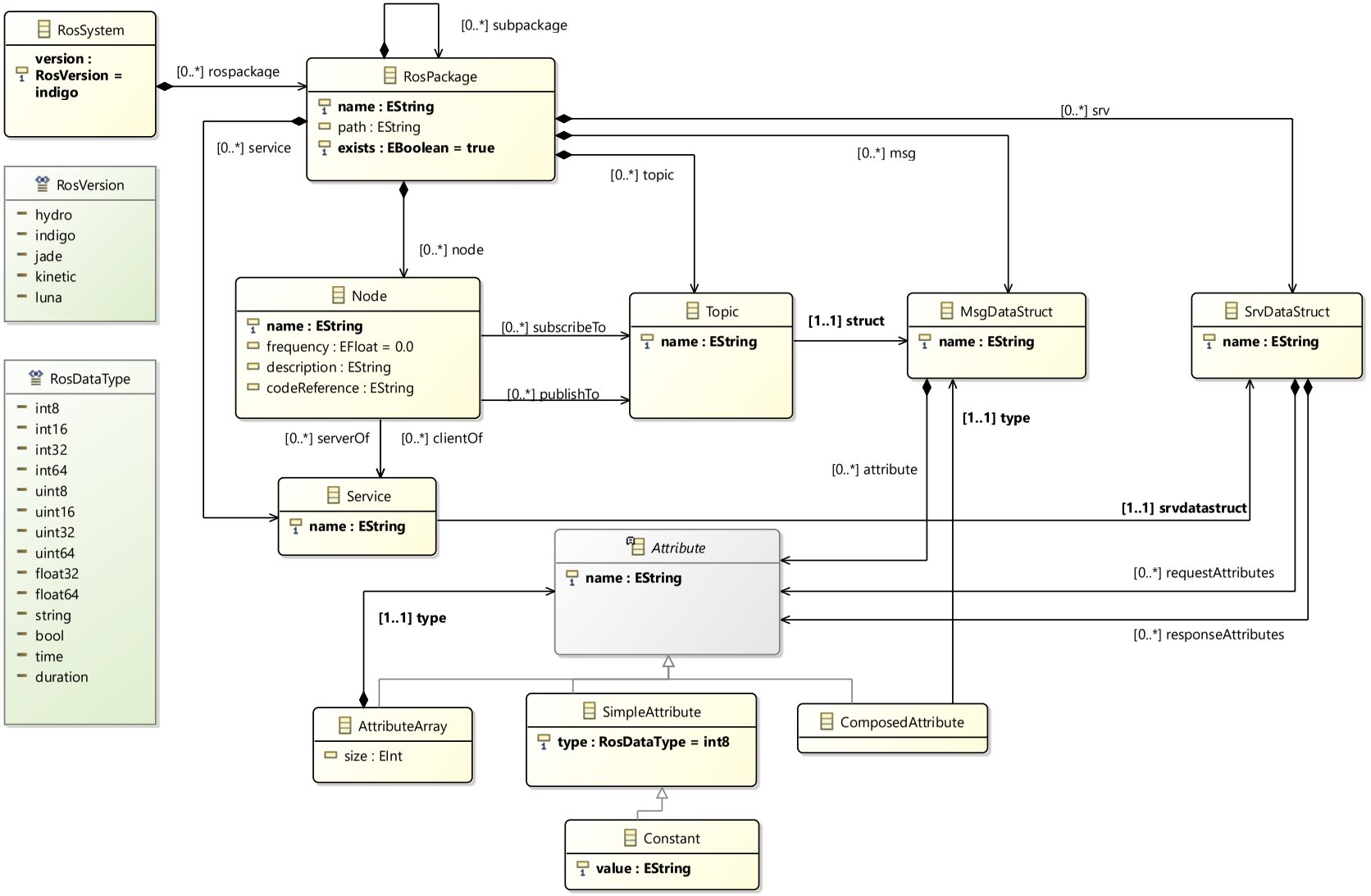

Figure 6.3: ROSModL Metamodel

on the current file system, and a boolean value indicating if it already exists (true) or if it requires to be created (false). A `RosPackage` instance may be composed of instances of `Node`, `Topic`, `Service`, `MsgDataStruct`, `SrvDataStruct`, and also other instances of `RosPackage` as sub-packages.

Each instance of `Node` class has a name, an execution frequency, and may have a reference to existing code in case it is already defined in the file system. It can provide or request access to any number of instances of `Service`, and publish or subscribe to any number of instances of `Topic`. The latter is defined by a name, and relate to a single instance of `MsgDataStruct` class. `Service` class is also defined by a name and is related to a single instance of `SrvDataStruct` class. While `MsgDataStruct` class has one group of `Attribute` instances, `SrvDataStruct` class has two groups of `Attribute`: one for request and the other one for response. Instances of `Attribute` class are defined by a name and one of these types: `Simple`, with a defined ROS variable type (with or without a defined value); `Array`, with a defined size and another attribute as its variable type; or `ComposedAttribute`, containing itself another `MsgDataStruct` instance as its type.

With this defined metamodel, it is now possible to use a code generator (e.g., Acceleo[2]) to automatically generate a compilable C++ code from any ROSModdL instance, since almost every information needed to generate the ROS system code is contained in the model. In fact, a transformation code has been written in Acceleo to execute this transformation. After running the transformation code directly on top of a ROSModdL instance, the generated C++ code serves as a skeleton for a ROS system, containing all files necessary to compile the system, including package files. The only information this metamodel does not address is that of the core algorithm inside ROS nodes: this is taken into account by the ROSProML metamodel.

6.2.2 ROS Process Modelling Language (ROSProML)

ROSProML can instantiate a general process being executed in a ROS system. A process here is defined by a functionality, a set of instructions executed in order. The metamodel is presented in Figure 6.4.

`Functionality` is the root class of this model. Its instances are characterized by a name and a frequency of execution. They are composed of (i) a single instance of `InternalBlock` class as its main block of code, (ii) a group of instances of `DataPort` as input and output variables, (iii) a group of instances of `ServicePort` and (iv) their related instances of `ServiceBlock` class.

It is important to notice that the frequency of execution is, in this case, a reference value. In the process of generating the code, for example, a `Functionality` instance will be referenced by an instance of ROSModdL, and this is the one responsible for determining the execution frequency. The lack of a defined frequency in a `Node` instance is what will make it

[2] `https://www.eclipse.org/acceleo/`

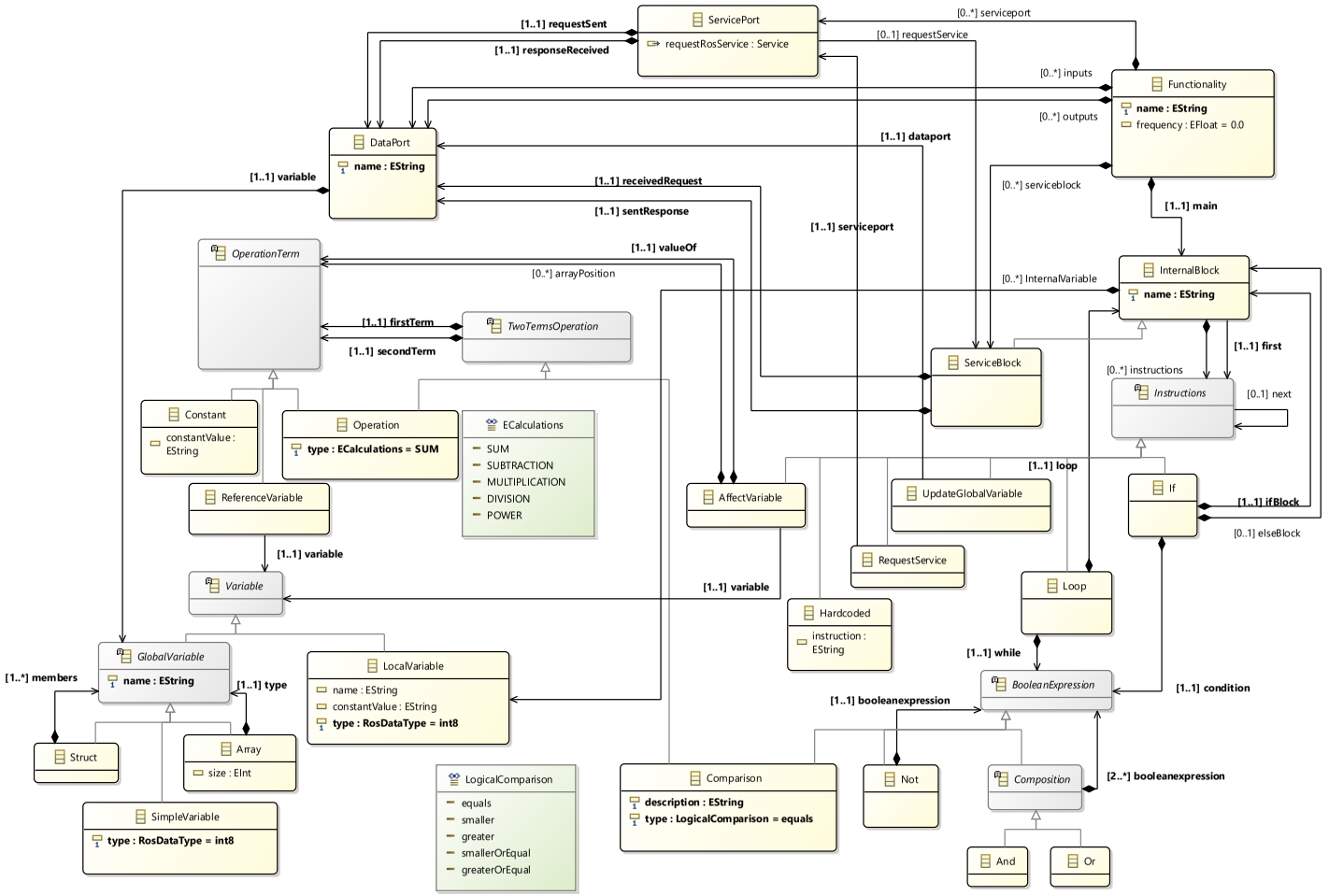

Figure 6.4: ROSProML metamodel

possible for the final execution frequency to be defined by the ROSProML's class attribute.

Also, `ServiceBlock` represents the implementation of a server for a ROS service – a Client-Server kind of communication. This means that their instances will provide an algorithm that will send a response to a specific `DataPort` instance after receiving an input from another `DataPort` instance. Complementary to that, a `ServicePort` instance, acting as a client, is what will request a response from a service implemented elsewhere.

`InternalBlock` is a class representing a block of code, defined by a name, that is composed of instances of `Instructions`. `Instructions` is an abstract class, where each of its concrete implementation is related to one or more variables of some kinds, according to the role it plays. Each kind of instruction will have a specific set of relationships with different features: for example, instances of `Loop` and `If` will each relate to a boolean expression, which in this case can be either a comparison (implemented by the class `Comparison`) between existing values (of a compatible ROS type) or a composition of comparisons made by logical operators. Each `Comparison` instance relates to two `OperationTerm` instances, in other words, to a constant, a variable or an operation containing its own pair of `OperationTerm` instances. (A textual description of the comparison was introduced as a Description attribute to ease the interpretation of the instance, but it is not relevant to the model itself.) On the other hand, the class `UpgradeGlobalVariable` is connected only to `DataPort`, representing the operation of exposing a value to a port. As another example, the class `Hardcoded` represents an instruction that had to be written manually, so the designer can have the possibility to use features that are not described by the metamodel. `ServicePort` class connects some of its elements with ROSModL entities, in order to allow the connection between the two DSMLs. By using this class, a `Functionality` instance can be connected to a `Service` instance from ROSModL.

With this definition, it was possible to implement an Acceleo code prototype that transforms the instances of this metamodel into C++ code.

6.2.3 ROS Mission Language (ROSMiLan)

ROSMiLan is based on behaviour-trees. Presented in Figure 6.5, has its root in the class `BlockOfConnectedFunctions`. This class is abstract, and its concrete heirs are the classes `Mission` and `Multitask`. Its instances have a set of at least one `MissionMember` instance, and they may have a set of `DataModules`, which connects to a `DataPort` from ROSProML, or to a topic from ROSModL.

Instances of `Mission` class are a named set of sequential members, inheriting from the class `BlockOfConnectedFunctions`, that can be executed in a ROS environment. It must specify one instance of `MissionMember` that will start the mission execution.

`DataModule` instances, responsible for transporting the values of variables from a module to another, have a defining name and must connect to at least one `DataPort` instance from ROSProML. `ExistingTopic` class inherits the characteristics of `DataModule`, but with an ex-

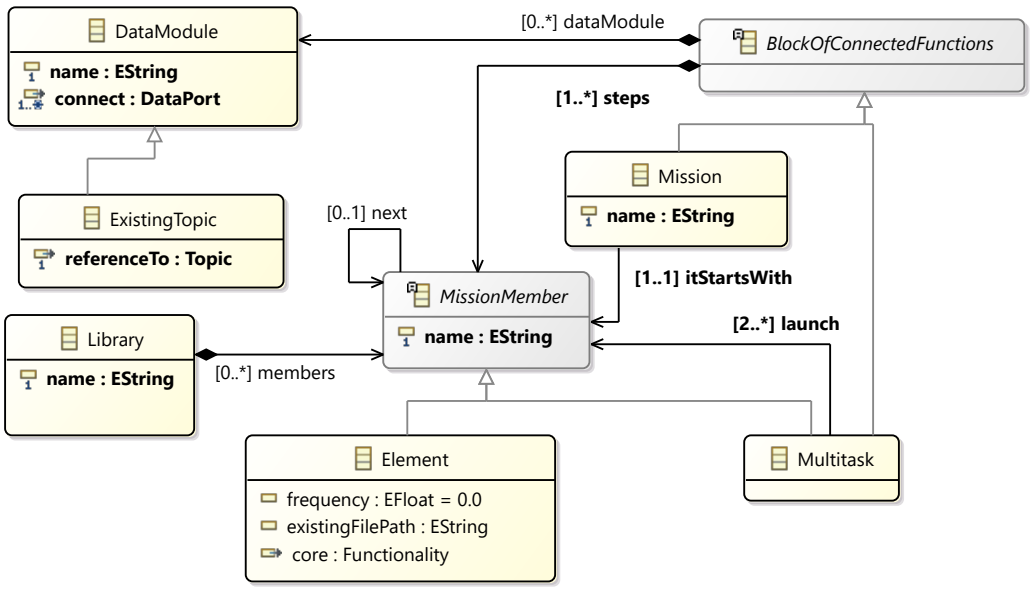

Figure 6.5: ROSMiLan Metamodel

tra reference to a `Topic` instance from ROSModL – therefore, it is supposed to be used when one wishes to use a ROS topic already instantiated in an external system. `MissionMember` instances are defined by a name and may be of type `Element` or `Multitask`. An instance of `Element` has an execution frequency and either a core functionality or the path to an existing algorithm in the file system. This functionality is another link to ROSProML. An instance of `Multitask`, on the other hand, simply connects to at least two instances of `MissionMember` that will start executing in parallel. (This means that an instance of `Multitask` can contain other instances of `Multitask`, in a limitless recursion, which allows the instantiation of as many parallel tasks as one wishes, while keeping the possibility of using a block of multiple tasks as a whole inside a sequential structure, with other blocks being executed before and after the referred block.) `Library` instances are named sets of unordered members (instances of `Element` or `Multitask`), which serve as a repository for sharing mission components between developers.

In order to understand the essence of this metamodel, it should be said that instances of `Mission` are used to generate code. Recalling that a ROSMiLan instance connects mission elements, it has been decided that it would do so by instantiating a new ROS system, which would contain the desired mission. Then, a ROSMiLan instance could be transformed in a ROSModL instance and, from it, have its code generated. This system has to contain at least an execution node, responsible for the first sequence of elements. When these elements are structured only in a sequential manner, there is no need to use more than one node, but every `Multitask` instance will require more nodes. Considering that each `Multitask` instance refers to a number N of "children" (elements referred to by a `launch` reference), this would represent the execution of N parallel tasks if all of them are `Element` instances. In other words, for the considered model, at least N nodes have to be in execution at a given moment.

To ease the implementation, the original node, containing the `Elements` that came before the `Multitask` instance, will be assigned to simply wait for the execution of all the "children" nodes to finish, so it can continue with the sequence of `Element` instances later. Hence, each `Multitask` instance adds N nodes to the system that already contained 1 node. To illustrate this, Figure 6.6 depicts an example of a hypothetical ROSMiLan instance, where each colour represents a different execution sequence, the bright red arrow points to the first task to be executed, brown arrows to the following one, and `Multitask` instances are connected to their "children". Figure 6.7 represents the corresponding ROS system, where each node appears as a rounded yellow rectangle with an arrow inside, representing its duration in time (the horizontal axis), tasks being executed are represented in blue arrows, idle tasks are represented in grey arrows, and start/ending signals are represented as vertical arrows.

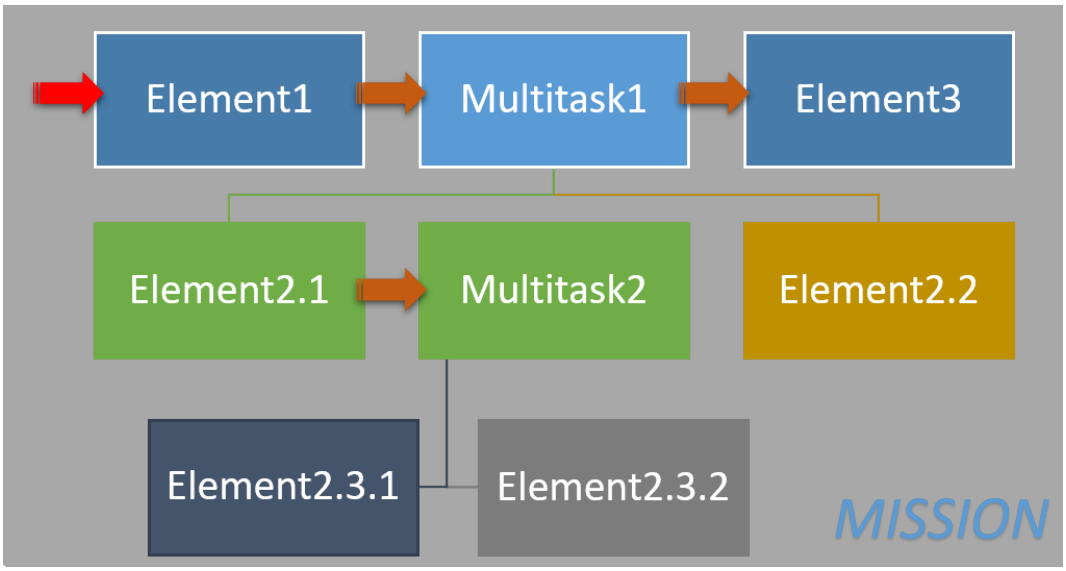

Figure 6.6: Example of a ROSMiLan instance with nested Multitasks

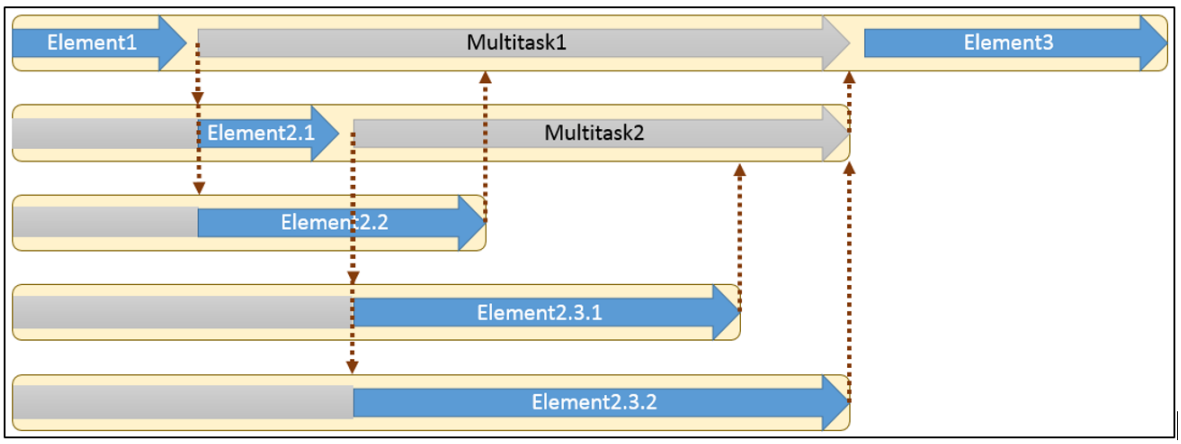

Figure 6.7: Corresponding ROS system regarding the ROSMiLan example of Figure 6.6

6.2.4 Connection between metamodels

The three metamodels mentioned before work in cooperation. Not only that, but ROSProML is dependent on ROSModL because of the reference to the class `Service`, and ROSMiLan is dependent on the other two because of the references to `Topic` (ROSModL), `DataPort` and `Functionality` (ROSMiLan) classes.

Due to the different objectives and contexts of use of the dependent metamodels, they will often need to have redundant information, such as the attribute Frequency in `Node` (ROSModL), `Functionality` (ROSProML) and `Element` (ROSMiLan). In the case of a Frequency attribute, for example, although the three different attributes define the same property of the modelled system (a task's periodicity), the context in which it is explicitly defined adds some information to this definition: in ROSModL, an explicit frequency of execution is a descriptive attribute made visible mostly to allow a better system analysis; in ROSProML, it means that the modelled task must imperatively execute in the specified time constraints e.g. due to physical limitations that would destabilize the system otherwise; in ROSMiLan, the frequency attribute would be an imposition of a certain system architecture that derives, for example, of a real-time analysis of the system's software and hardware.

6.3 Proof of concept

RoBMEX is used here as a proof of concept, envisaging as a first step the generation of compilable (free of syntax errors) code.

6.3.1 Tooling

We have used the capabilities provided by Eclipse Modelling Framework plugin to set up the metamodels of the three DSMLs as well as the tree editors dedicated to the instantiation. The development of another user-friendly graphical concrete syntax is in progress. This latter is based on the Sirius[3] tool.

Figure 6.8 shows the structure of the RoBMEX tool and the existing relationships between different components of the developed plugin, which expresses the proposed DSMLs. Furthermore, we have developed a C/C++ code generator. The implementation of this latter is based on Java programming language and Acceleo.

The metamodels and transformation tools are available online[4].

[3] `https://www.eclipse.org/sirius/`
[4] `https://github.com/lias-laboratory/robmex`

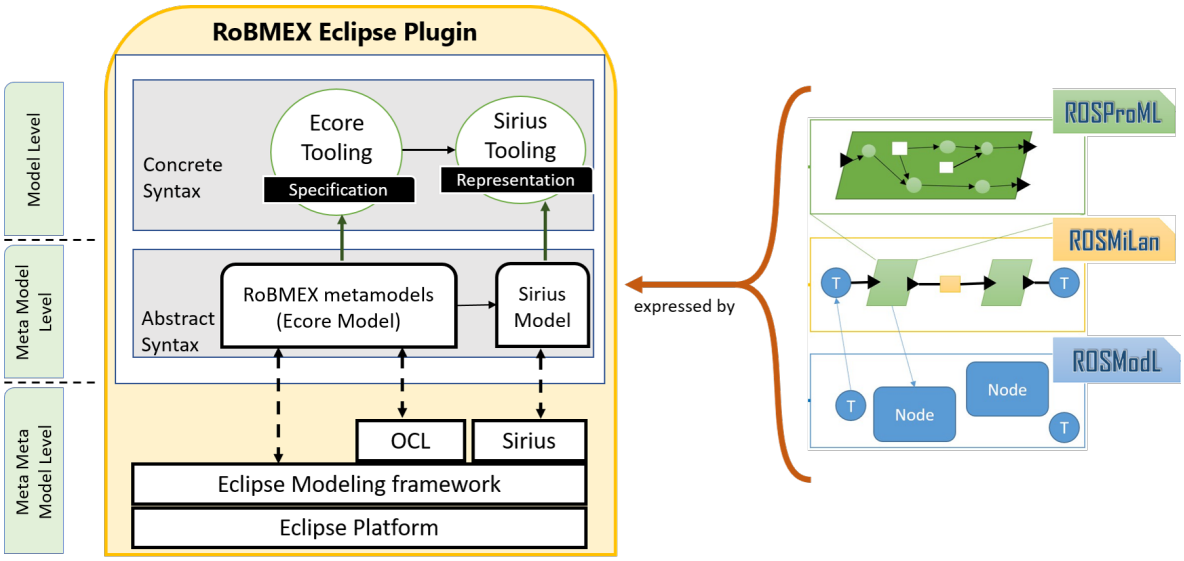

Figure 6.8: Structure of the RoBMEX editor

6.3.2 Example

In order to demonstrate the capability of the proposed framework, a simple scenario was conceived with two variants. In this scenario, a flying drone equipped with a proximity sensor needs to execute a sequence of activities:

1. Arm the motors;
2. Takeoff;
3. Move (towards an objective);
4. Land.

This sequence is treated as the drone's mission that we want to implement, and it is represented in Figure 6.9. The Move step, that takes place after the step Takeoff has finished, is itself a mission, where the drone has to execute two parallel loops: setting the direction of movement, and evaluating the proximity to any obstacle. The latter will determine, using static parameters and drivers to the sensor, if the distance to a detected obstacle is safe or not. In case it is not safe, a signal is sent to the former so that an "avoid" mechanism can take place, and the mechanism will be sustained until the received signal returns its status to "safe". The Land step will be executed after the reception of the relative command from the pilot, which will finish both the parallel tasks simultaneously.

In the first variant of the case, the mission will be composed using modules that communicate directly with the drone's drivers. In a second variant, the modules have to communicate with an autopilot that only supports MAVLink communication and hence will need a kind of translation. For this, an existing ROS system called MAVROS [Erm18] will be used. This system is a MAVLink extendable communication node for ROS with proxy for GCS, and it is responsible for converting data from inside ROS systems (in topics and services) into

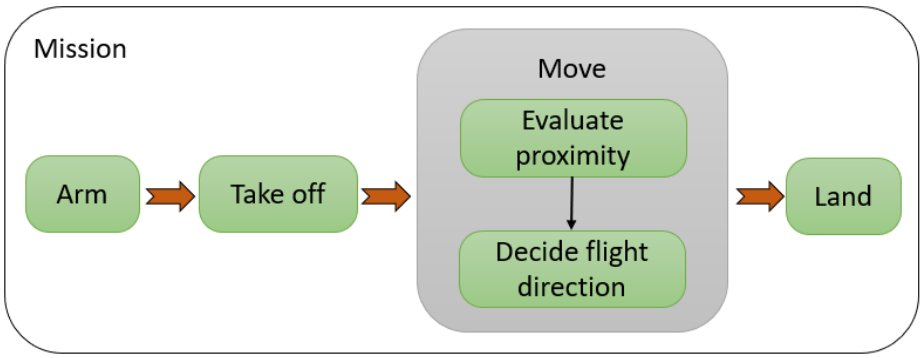

Figure 6.9: Representation of the designed mission

MAVLink messages and send them to an autopilot software, and vice-versa. Using the ROS command `rqt_graph` after initializing MAVROS, it is possible to get a graphical representation of the MAVROS node in execution, as well as the topics and services it relates to. From an excerpt of this representation, in Figure 6.10, the MAVROS node is seen as an elliptical form, while some of its topics and services are represented within rectangles. The arrows indicate the sense in which data flow: when an arrow goes from MAVROS to a topic, MAVROS publishes to the topic, while an arrow in the other sense means MAVROS subscribes to the topic.

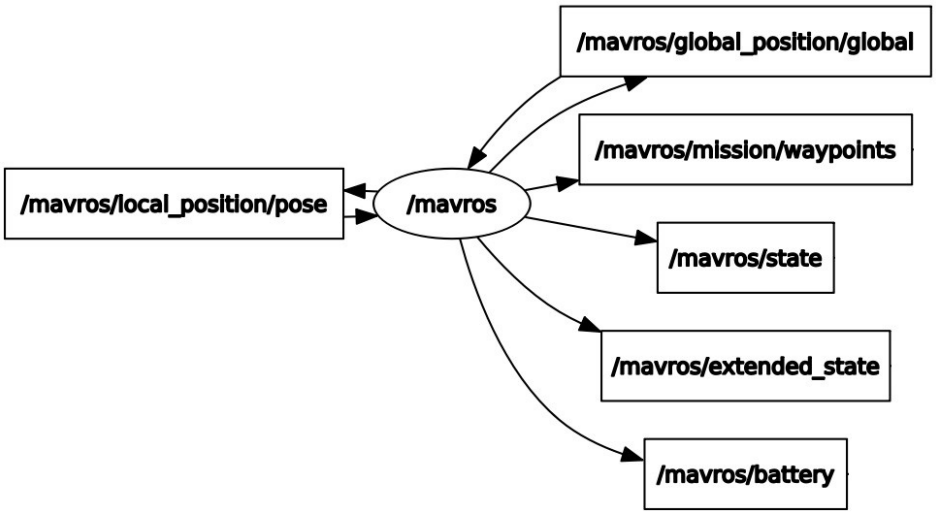

Figure 6.10: Excerpt of the ensemble of MAVROS's topics and services, extracted from `rqt_graph`

6.3.2.1 First case: Direct communication to drivers

As a first variant, the mission is instantiated as in Figure 6.11 using ROSMiLan. Each `Element` instance relates to a specific C/C++ file that contains the algorithm for the desired

behaviour – either pointing to an already existing one at the location defined by its attribute "existingFilePath", or automatically generating the file in case it is non-existent. The instance of `DataModule` represents the value that is transmitted from "EvaluateDistance" to "DecideMotionDirection". The latter communicates directly with the autopilot.

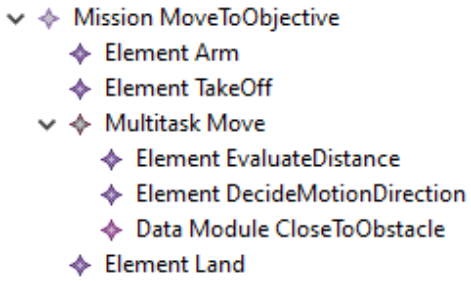

Figure 6.11: ROSMiLan instance of the designed mission, with no MAVLink communication

In this case, the referred C/C++ files can be created as instances of ROSProML. As an example, we instantiate the basic functionalities of the elements of Move, seen in Figure 6.12 (measures a distance, compares to a threshold and outputs the boolean result as "DecideMotionDirection") and Figure 6.13 (sends a positive or a negative value to motors depending on the value of the boolean input, representing whether there is or not an obstacle ahead, pointed by "EvaluateDistance").

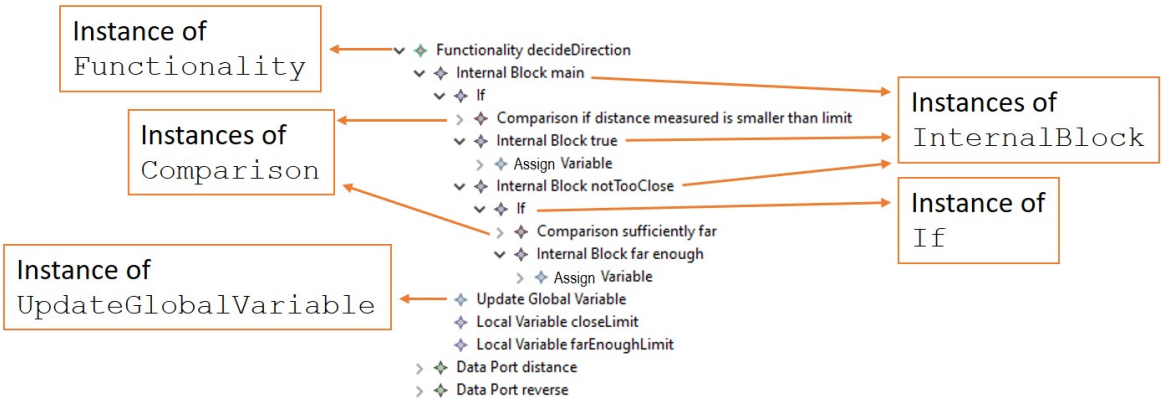

Figure 6.12: Excerpt of the ROSProML instance DecideDirection

This ROSMiLan instance could, in theory, be transformed into a ROSModL instance according to the principles shown in section 6.2. Then, from the ROSModL instance, the code can be generated. Using a sample of a transformation model written in Acceleo for automatically generating code, it was possible to generate compilable (i.e., absent of compilation errors) functionality codes from the ROSProML instances, as presented in Listing 6.1 and Listing 6.2.

It is clear from the generated codes that they would not work in a real environment, since publishing to the global variables was not addressed. This is purposeful, once the objective was only to do a sample of code generation: the complete version would require considerable

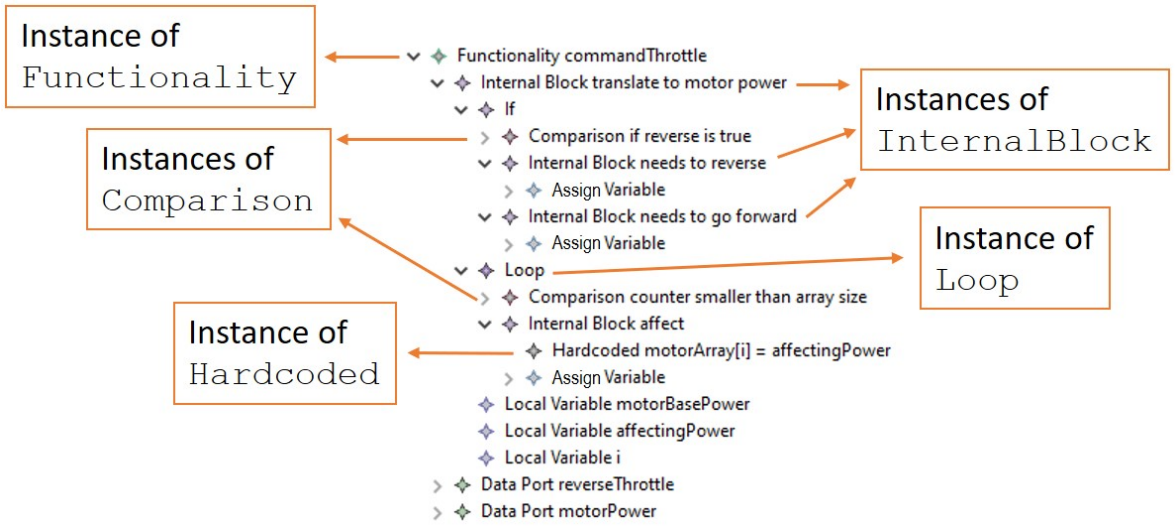

Figure 6.13: Excerpt of the ROSProML instance CommandThrottle

amount of work. Hence, we chose to focus on the most basic elements of the metamodel, which excluded dealing with global variables. Nonetheless, we do not see any constraint to consider them for automatically generating code.

6.3.2.2 MAVROS communication

As a second variant, the same mission is instantiated but with a "DecideMotionDirection" that relates to an existing MAVROS topic, as is shown in Figure 6.14. For this, MAVROS had to be instantiated so that its elements could be accessible by the model, as seen in Figure 6.15.

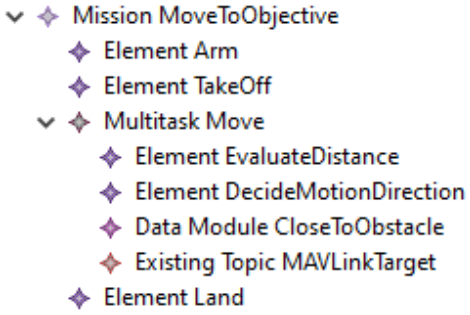

Figure 6.14: ROSMiLan instance of the designed mission, with MAVLink communication through MAVROS topics

Using the code generation algorithm developed for ROSModL, it was possible to generate a compilable (i.e., absent of compilation errors) C++ code which contains the structure needed to work with ROS, though lacking the behaviour of the generated functions, which were not determined. An excerpt of the generated code can be seen in Listing 6.3.

Listing 6.1: Generated code for the ROSProML instance DecideDirection

```
// AUTOMATICALLY GENERATED FILE FOR: decideDirection

// Structs:



// Global Variables:
        long double  m;
        bool  value;

void main_decideDirection() {
                short int  closeLimit  = 10 ;
                short int  farEnoughLimit  = 20 ;
         if ( (m)  <  (closeLimit) ) {
                        value = true ;
              }  else {
                        if ( (m)  >  (farEnoughLimit) ) {
                                value = false ;
                        }
              }
         /* publish to reverse */
}
```

Listing 6.2: Generated code for the ROSProML instance CommandThrottle

```
// AUTOMATICALLY GENERATED FILE FOR: commandThrottle

// Structs:



// Global Variables:
        bool  reverse;
        long double  [4] motorArray;

void main_commandThrottle() {
                long double  motorBasePower  = 100 ;
                long double  affectingPower ;
                long long int  i ;
         if ( (reverse) == (true) ) {
                        affectingPower = (motorBasePower) * (-1) ;
                } else {
                        affectingPower = motorBasePower ;
                }
         while ( (i) < (4) ) {
                        motorArray[i] = affectingPower ;
                        i = (i) + (1) ;
                }

}
```

Listing 6.3: Excerpt of the generated code for the ROSModL instance of MAVROS

```
#include "mavros_msgs/WaypointSetCurrent.h"
#include "mavros_msgs/CommandBool.h"

#include "mavros/gen_node_mavros_body.h"


// Callbacks to subscribed topics:

void callback_rc_override(const mavros_msgs::OverrideRCIn::ConstPtr& msg)
{
        // Code
}

void callback_setpoint_position_local(const geometry_msgs::Pose::ConstPtr& msg)
{
        // Code
}
```

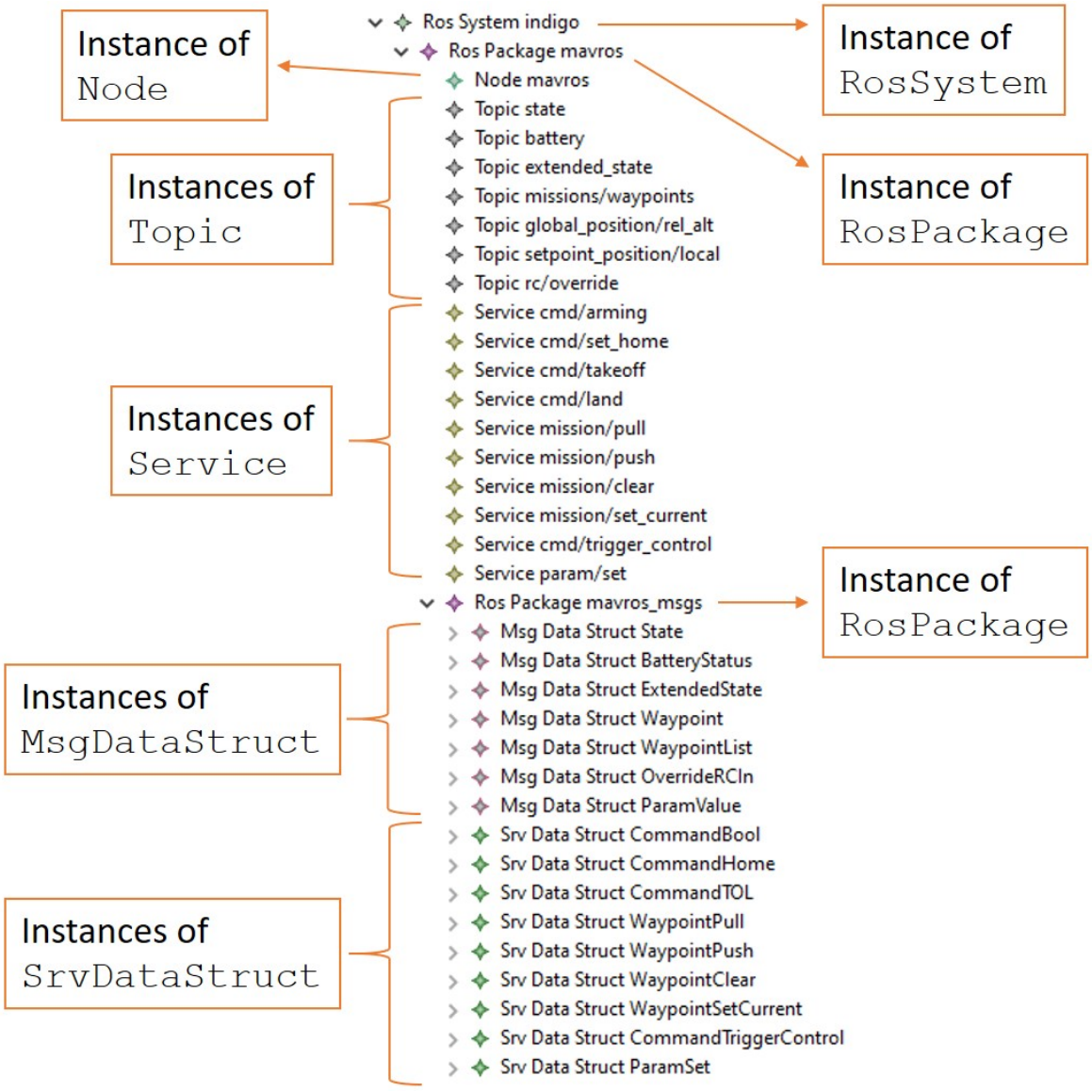

Figure 6.15: Excerpt of a ROSModL instance representing MAVROS

6.4 Conclusion

In this chapter, current open source technologies involving the conception and execution of drone behaviours, or missions, were discussed: ROS and MAVLink. Besides presenting a certain degree of modularity and reusability, they also present limitations, especially regarding the time and effort taken by non-experts to get the necessary know-how to use them for their purposes.

Faced with the interest in reducing these efforts, we proposed the RoBMEX approach, composed of three complementary DSMLs. To model ROS , we presented ROSModL, which is the basis of the framework for being capable of describing the structure of a general ROS system. Each of its execution nodes can be modelled using ROSProML, a DSML designed for algorithm experts that can represent the process of transforming input signals into the desired outputs using basic operations, function calls, comparisons and logic blocks. The connection between execution nodes can be instantiated using ROSMiLan, the DSML responsible for designing the mission itself by giving a non-expert end-user a simpler language that doesn't require any expertise in terms of the drone technologies.

Then, an example of a ROSModL instance was made, with a variant that is based on the existing system MAVROS and another that is completely independent of other technologies. Some of MAVROS's elements were successfully represented using ROSModL, and could then be referenced by an instance of ROSMiLan (a model of a system that intends to communicate with an executing MAVROS node). Two examples of ROSProML instances were also made based on hypothetical tasks, and had most of their code automatically generated in C/C++.

We can now compare the developed framework with some of those presented in the previous chapter. Table 6.1 summarizes the characteristics of the discussed alternatives to instantiating drone missions.

Table 6.1: Summary of mission implementation technologies.*

Ref.	Method	I.x0	D.	P.	S.C.	C.C.G.	ROS
-	**Human at GCS**	-	Yes	-	-	-	-
-	**Rewrite autopilot**	Yes	-	-	-	-	-
[Miy+19]	**RoboChart**	Yes	-	Yes	Yes	In dev.	-
[FLY19]	**FlyAQ**	-	Yes	-	-	Yes	-
[Rob19a]	**RobMoSys**	Yes	Yes	Yes	Yes	-	Yes
[Wen+16]	**ReApp**	Yes	Yes	Yes	Yes	-	Yes
[Lot+17]; [Bru15]	**SmartSoft**	Yes	Yes	Yes	Yes	-	Yes
[Dho+12]	**RobotML**	Yes	Yes	Yes	Yes	Yes	-
[BRI14]	**BRIDE**	Yes	Yes	Yes	Yes	Yes	Yes
-	**RoBMEX**	Yes	Yes	Yes	Yes	In dev.	Yes

* *Ref.* stands for Reference, *I.x0* for Independent of Initial Position, *D.* for Dynamic (able to change mission at run-time), *P.* for Parallel, *S.C.* for Share Components, *C.C.G.* for Complete Code is Generation, *ROS* for Integration of ROS, and *In dev.* for In development.

As possible improvements, firstly RoBMEX could be adapted to generating ROS2 systems instead of ROS systems with a relatively low effort, due to the similarities between the two technologies. Also, RoBMEX can be further tested through different use cases in order to obtain a wide range of choices for reusable libraries and missions. In addition, it is possible to align the framework to use tools of real-time analysis, which can evaluate the schedulability of the system and calculate the response time for its tasks, in the light of [FH20]; [Cas+19]. This would allow the tool to have a library of drone mission components that would also be evaluated in real-time aspects (e.g. deadlocks and missing deadlines) for validating the design as soon as it is modelled, such as in [Abb+20]. Our work can also be extended in order to design swarm missions. In this case, our work should be coupled with [Boz+19] in order to take into account the collective adaptation problems, especially in run-time mode. Furthermore, additional concrete syntax can be developed, especially the graphical ones, allowing to easily instantiate drone missions. Also, RoBMEX is downloadable to be shared with the community under the LGPL licence [LIA21].

Uniting the code created by the transformation over ROSModL and the one over ROSProML, it will be possible to generate a complete compilable and executable file. This shall be the mission of the code generation for ROSMiLan models, which is still in development. It is important to note that the code generation can have other targets – not only C/C++ but also Python, and if no ROSModL instance is used, other generators can be implemented to work on ROSProML and ROSMiLan instances to create code in other languages used in embedded systems, such as Ada.

Chapter 7

Reference Architecture

In the previous chapter, we proposed an approach to customize drone missions using a DSML, RoBMEX. It allows the specification of missions without changing the internal aspects of the drone, keeping a validated core system unchanged.

However, for some situations, using only the interfaces provided from the autopilot supplier is not enough: a deeper customization must be made, effectively changing the core functionalities of a drone. In that case, it is preferable to use Modifiable Off-The-Shelf (MOTS) components, which are similar to COTS but are open to specific changes to its internals. MOTS components allow the customization of developed systems to the developers' needs, while also providing the possibility of reuse of most development made within their domains.

Such an approach can be adopted with the use of a *reference architecture*. The reference architecture serves as a basis, a set of roles that must be played in order for the system to work properly. With such an architecture, interchangeability between different implementations is made possible, paving the way to safe customization and reusability of drone modules.

In addition, the analysis of such embedded systems is facilitated, promoting safety and security inspections that improve their trustworthiness. A better comprehension of the whole system by anyone who plans on changing its code is also encouraged with the compliance to a reference architecture.

In this chapter, we identify what are the currently most used architectures in open-source drone autopilots, reverse-engineering a reference architecture capable of contemplating all these autopilots but also its derivations. With such, we propose a general reference architecture.

7.1 Approach

When developing a new reference architecture for drone autopilots, two different approaches could be followed. The first is developing a new architecture from scratch, imposing new patterns and prescribing methodologies that autopilot developers should use. However, this is not sustainable and would not catch the interest of the main concerned stakeholders, because:

- Existing off-the-shelf autopilots, such as Paparrazi, PX4 and Ardupilot, have been developed and tested for hundreds of thousands of hours thanks to thousands of contributors

and users. Developing from scratch a new autopilot would require such a huge effort and create a useless concurrence with already existing products.
- New embeddable systems on chips, sensors, actuators and operating system versions that could be used to run an autopilot are released on a weekly basis. Porting an existing autopilot to new platforms, sensors and actuators also requires a large community of contributors to make a software product living and scalable.
- Existing autopilot projects have an enormous amount of cumulated work put into their code, and this can be seen by the number of files and total code lines of each autopilot project. PX4, for example, has 2015 C files, which add up to more than half a million lines of code, and Paparazzi has 526 C files, adding up to almost two hundred thousand lines of code. Regarding C++ files, PX4 has more than a thousand, which contain in total almost three hundred thousand lines of code. If we consider also makefiles and configuration files, they can contribute with tens of thousands of lines of code. Yet, each autopilot generated by the target compilation hardly uses 10% of the files.

The risk of obsolescence of such an approach is therefore too high, and the chances for adoption are too low. Therefore, we decided to adopt another approach, one that allows the reference architecture adoption to be as effortless and safe as possible.

By "effortless", we mean that, when designing a new custom component, the developer should not need to have a complete expertise of the target autopilot. Otherwise, safety problems, cost in time and resources, problems with future evolutions of the autopilots and versioning would become large obstacles for the integration. When integrating a custom component, the developer should be able to quickly identify by what means the component can be integrated. They should be able to perceive how it can gather its inputs and how its outputs can impact the behaviour of the autopilot. During this process, the possible occurrences of race conditions should also be clearly visible.

By "safe", 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.

Hence, by analysing current architectures and extracting common characteristics, a single reference architecture can be described, to which the studied autopilots already comply. The next sections present the current architectures and, later, the common perspective uniting them.

7.2 Current Architectures

Knowing that current open-source drone autopilots have their core (movement control functions) deployed on a single processor (even if it is multicore) in a single board, we can focus on three perspectives to analyse current architectures:

1. **Functional architecture:** consists of the subdivisions of functionalities, how the sys-

tem behaviour is decomposed into smaller functions.

2. **Software architecture:** shows how the application code is subdivided, the modules used to compose the larger system.
3. **Real-time architecture:** regards the execution threads and implementation details that compose the system, implementing the functional architecture.

7.2.1 Functional Architecture

With respect to the functional architecture of drone autopilots, the three autopilots studied in this thesis are decomposed into the same core functions. The movement of the drone is controlled by a cascaded control, such as shown in Figure 1.8. The cascaded control is further detailed in Figure 7.1 and summarized as follows.

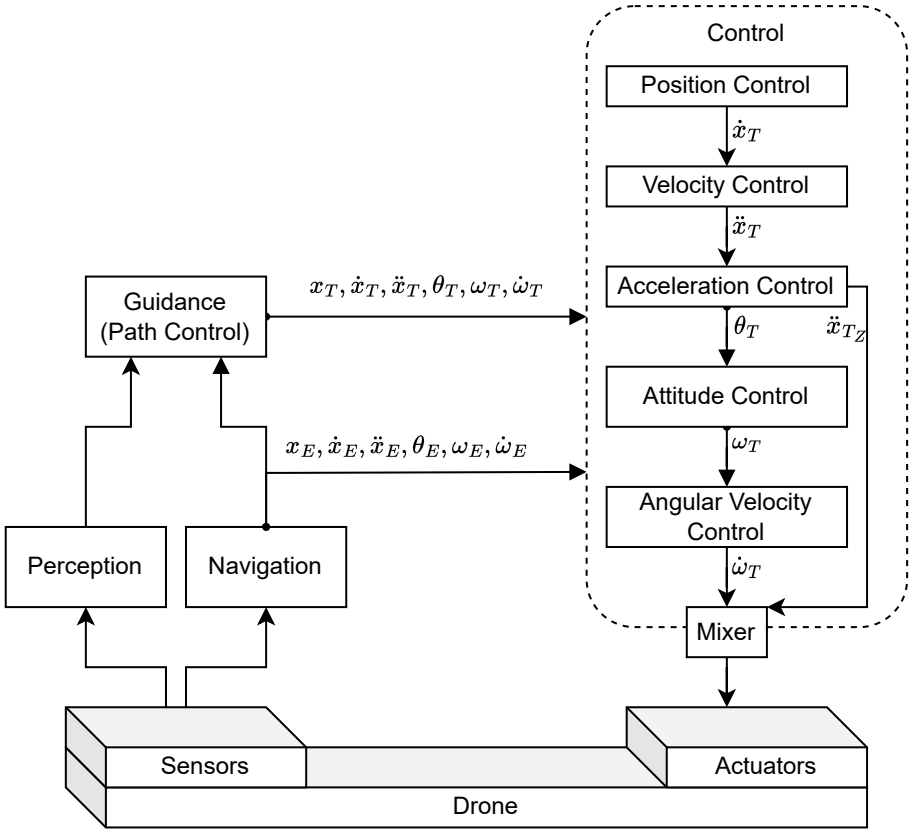

Figure 7.1: General functional architecture in an autonomous drone

In Figure 7.1, physical components are represented by rectangular cuboids: sensors, actuators and the drone itself. The functions responsible for the autonomous flight are represented as rectangles.

Sensors send data to the Navigation and the Perception functions. The Perception function, not always present, is responsible for interpreting data about the surroundings of the drone, so it can identify entities which are close. The Navigation function, required in au-

tonomous drones, interprets data to identify the current position and orientation of the drone, translating it into a state vector, composed itself of the following vectorial quantities (in 3 dimensions): the estimated position x_E and its derivatives (estimated velocity $\dot{x}_E$ and estimated acceleration $\ddot{x}_E$), as well as the estimated angular position θ_E and its derivatives (estimated angular velocity ω_E and estimated angular acceleration $\dot{\omega}_E$). This function is often called State Estimation, and it might include filters such as simple low-pass filters or a more sophisticated Kalman filter, for example.

Once the state vector is defined, its information is made available to the Control function group, represented in Figure 7.1 as a dashed rounded rectangle. The control functions are responsible for maintaining the drone's stability. The lowest-level control function is responsible for stabilizing the drone's angular velocity, and due to the small time constant of the control system, it must be executed in a high frequency in order to guarantee its stability. From the estimated values of angular velocity and acceleration, and from a desired angular velocity ω_T, it calculates a target angular acceleration $\dot{\omega}_T$ at a rate of at least 400 Hz, often arriving at 1 kHz and even 3kHz. Then, $\dot{\omega}_T$ is used, for example in a multicopter, by the Mixer to calculate the power needed by each individual rotor.

On top of the Angular Velocity Control function, the orientation (attitude) of the drone is stabilized as follows: the estimated value is compared to a target value θ_T, and a desired (target) angular velocity ω_T is then calculated and made available to the Angular Velocity Control function. These calculations happen at the same rate as the Angular Velocity Control function. On top of the Attitude Control, the Acceleration Control function calculates the target attitude and vertical acceleration $\ddot{x}_{T_Z}$ from their estimated values and the target linear acceleration $\ddot{x}_T$. This function, however, can operate at a much lower rate than attitude-related functions, due to a larger dynamic time constant in the linear system. A similar cascade control also operates for the linear position: the target acceleration is calculated by the Velocity Control, basing itself on estimated values for the velocity and a target velocity $\dot{x}_T$, which is itself calculated by the Position Control, which relies on the estimated value for the position and a target position x_T.

The position target itself can be given by an operator (the pilot) or by an autonomous Guidance system. The Guidance is responsible for managing the path of the drone: current targets and new ones, retracing routes, etc. It is based on the outputs of the Perception and Navigation functions, as well as inputs from external systems (a pilot or a GCS), and it can determine specific target values for the control functions.

Other functions, such as payload management, communication and logging, have been hidden since they are not critical for the drone movement. However, they recover data from the autopilot's core, and might influence the decisions made by the Guidance function depending on the mission.

7.2.2 Software architecture

The open-source projects analysed in this thesis are, in fact, autopilot generators. Since each autopilot instance is a specific mission logic implemented in specific hardware which is connected to a specific drone (with its particular mass and thrust distribution), the projects actually generate a distinct autopilot for the desired target. Such an approach allows these projects to be very flexible due to the high degree of customization. This characteristic is only achieved by a layered software architecture that abstracts implementation details to a common framework.

In order to be compatible with distinct pieces of hardware, they contain a Hardware Abstraction Layer (HAL) as an important part of the code. It provides a common base for the autopilot modules, making them independent of the target and hiding the target-related complexity with its respective HAL glue. This approach is represented in Figure 7.2.

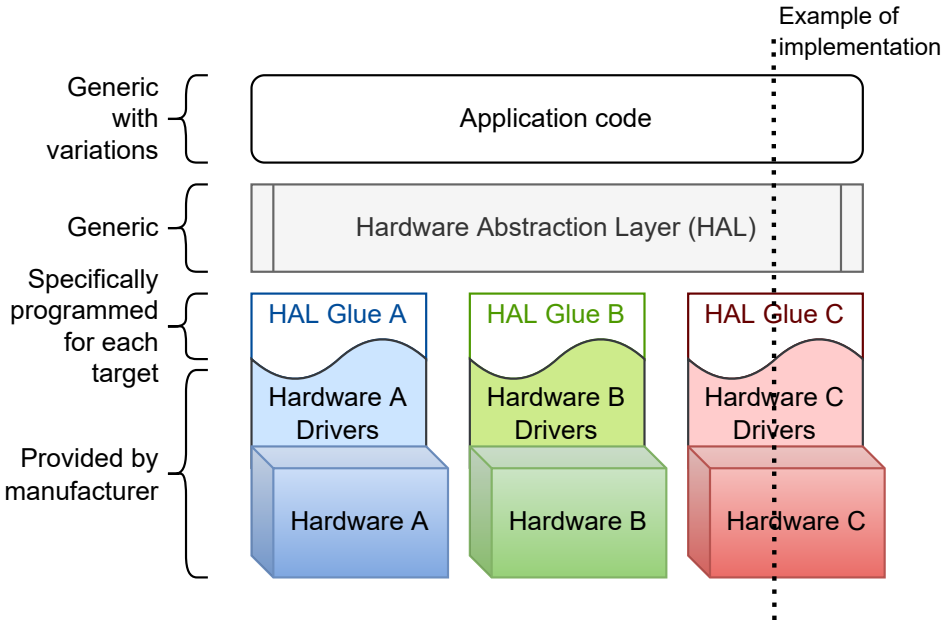

Figure 7.2: Representation of the Hardware Abstraction Layer (HAL)

In addition, so the application code (the autopilot) can be customizable, they are composed of smaller and interchangeable modules. Some of them use a middleware, since it abstracts communication details, allowing each module to contain solely the information relative to its own algorithm.

7.2.3 Real-time Architecture

ArduPilot, PX4 and Paparazzi rely on cyclic schedulers to run the autopilot's core algorithm. Even though some implementations might contain parallel threads, the movement functions are implemented by a single thread in the form of a cyclic scheduler (or two in the case of PX4:

one with lower frequency for the position control stack, and another with higher frequency for the angular control stack).

This choice can be explained by two aspects. Firstly, the cyclic scheduler approach is already validated by a large amount of flight hours that have been taking place throughout drone history. Secondly, implementing thread-safe architectures is not a trivial task, and it might even end up creating more problems than it might solve (race conditions, thread starvation, etc.), which would increase the rate of possible catastrophic consequences.

Yet, when the target architecture supports multithreading, some functions are preferably implemented by parallel threads, such as the communication with I/O ports, logging, and some payload management. The use of threads in these cases presents more advantages, since potential thread-related problems will not affect the autopilot's core, and the blockage of such threads (either by malfunctioning peripherals or by improper coding) will not propagate to the high-priority control thread.

Table 7.1 summarizes the characteristics explored in each autopilot in the current section. Note that, although there exists no native interoperability between ArduPilot and ROS, or between Paparazzi and ROS, there are freely available bridges between ROS and MAVLink such as MAVROS.

Table 7.1: Recall of the comparison between open source autopilots for drones

Autopilot	**PX4**	**ArduPilot**	**Paparazzi**
Cascade Control	Yes	Yes	Yes
Modular functions	Yes	Yes	Yes
Flight modes	Yes	Yes	Yes
RTOS / Parallelism	NuttX	- / ChibiOS	- / ChibiOS
Middleware	uORB	-	ABI
Thread-safe Middleware	Yes	No	No
Communication Protocol	MAVLink	MAVLink	PPRZLink
Interoperability	ROS, RTPS (DDS)	-	-
Licence	BSD	GPL	GPL

7.3 General Architecture

7.3.1 Software Architecture

In order to be compatible with distinct pieces of hardware, a drone autopilot must contain a HAL. The HAL provides a common base for the application code, making it independent of the target and hiding the target-related complexity with its respective HAL glue. This approach is represented in Figure 7.3.

In addition, so the application code must be composed of smaller and interchangeable modules, as it needs to be customizable. Each module must interact with each other through

a defined interface, one which allows the substitution of one module by an equivalent one. This can be accomplished with the use of a middleware, since it abstracts communication details, allowing each module to contain solely the information relative to its own algorithm and, thus, become agnostic and independent of other modules.

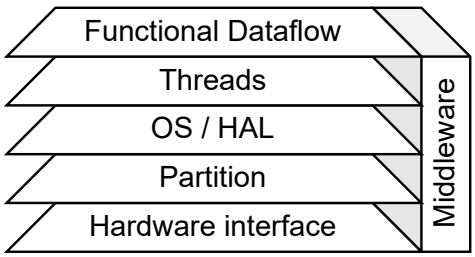

Figure 7.3: Layered architecture of an autopilot

7.3.2 Real-time Architecture

The thread architecture of an autopilot should follow the design choices that have been proven to work. For instance, a single thread (or, at most, two threads, as in PX4) must be used for the autopilot's core (GNC), using a cyclic executive. This design has been validated with thousands of hours of flight by the several drones running on PX4, Paparazzi and ArduPilot. The single possible subdivision should be the separation between the high-frequency control loops (attitude control) and the low-frequency ones (position control). Increasing the subdivisions would promote the appearance of thread synchronization issues.

Other threads can be used to deal with lower priority tasks, such as input and output device handling (I/O), logging, trajectory planning, etc. Nonetheless, their communication must be thread-safe, with well-defined strategies to avoid any kind of thread synchronization problem. A well adapted middleware can provide the necessary communication infrastructure.

The aforementioned architecture is represented by Figure 7.4. However, if the hardware does not support threads, or if no thread-safe communication can be guaranteed, a single-thread structure can be used. An example is given by the Paparazzi autopilot, discussed in section 7.5.

7.3.3 Functional Architecture

The proposed functional architecture is a set of roles that can be implemented. Every role is optional, but their presence and complexity are directly related to the autonomy of the drone. Such roles are explained in this section.

This functional architecture is based on Figure 7.5, extracted from [Ken12b]. RUAV stands for Rotary UAV, and RUAS, Rotary UAS. The flight control is the lowest-level of drone autonomy, necessary for every higher level of autonomy. Navigation (data acquisition and

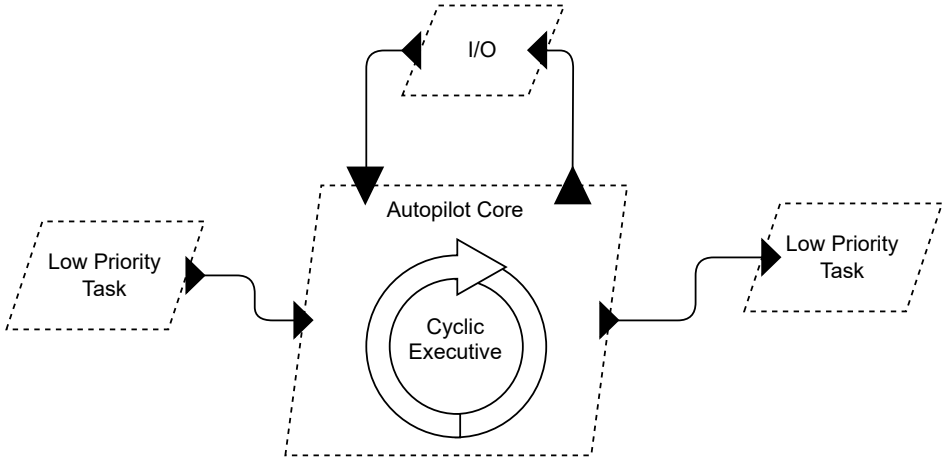

Figure 7.4: Real-time architecture example

interpretation) is responsible for a higher level of autonomy, a level that can vary depending on its functionalities, as seen in the figure. Guidance is the module responsible for the highest autonomy level, which can also vary depending on its functionalities.

For the general architecture, some of the functionalities assigned to the Guidance module in Figure 7.5 have been factorized in their own modules, since they are less important regarding the control loops and can be executed at much lower frequencies. Other functions have been highlighted, since they appear in most implementations. Also, the software bus is highlighted due to its fundamental aspect in modularity and safety analysis. It can represent simple global variables and a non-thread-safe internal middleware such as in Paparazzi, or a thread-safe middeware as uORB in PX4, with native bridges to external DDS systems. The proposed functional architecture is represented in Figure 7.6.

As the GNC module is the autopilot's core and, from a functional point of view, the most critical section of the embedded software, it must have a privileged communication with the drone's actuators. Other modules should limit its communication to the interface provided by the Guidance module, at most updating the current drone's target (except for the pilot override, a safety measure in case the software fails to properly execute its designed mission).

The Payload Management module in a drone autopilot is responsible for managing the payload carried by the drone during flight, intimately related to the drone's mission. In case the payload is a physical object that should be picked or dropped during the flight, the module should also make real-time adjustments to the drone's flight parameters to maintain stability. Otherwise (e.g., camera or special sensors), the module should only act on the carried object according to the specific moment of the mission or to external commands it receives. It can have a direct communication with the payload.

The Mission Management module can account for factors such as weather conditions, obstacles, and battery life in order to manage and adapt the current mission of the drone, optimizing the drone's flight path to ensure the mission's success. It must keep the Guidance

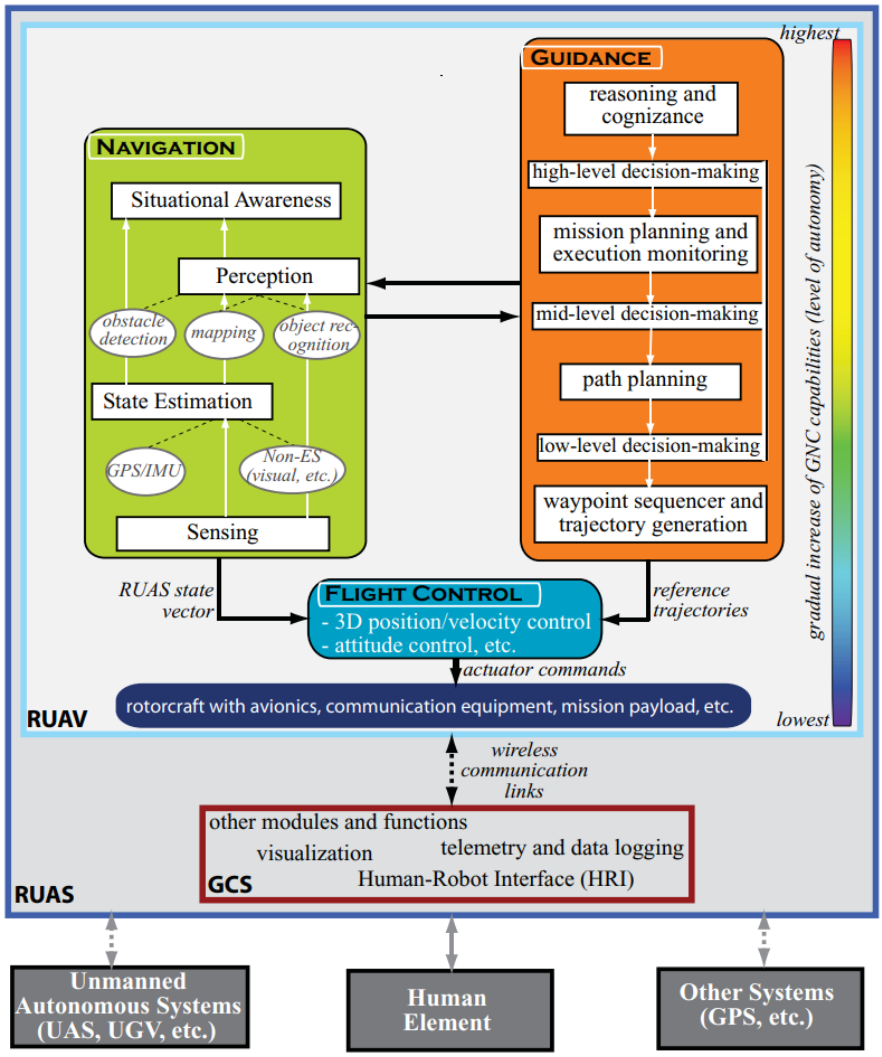

Figure 7.5: Recall of Guidance, Navigation and Control functions of an autopilot, and their relation with its autonomy [Ken12b]

module updated with the current path parameters.

The Health Management module is responsible for monitoring the health of the various subsystems of the drone (e.g., processor, batteries and sensors), detecting any anomalies or failures that may occur during flight. Additionally, it can provide valuable feedback to the pilot, GCS and other connected systems in the network, enabling them to identify and address issues before they lead to more significant problems.

The Data Management module collects and stores data about the drone flight, such as altitude, speed, GPS location, battery levels and other sensor readings, in addition to received messages and commands. The module can also record video and images captured by the drone's cameras during flight. The collected data can then be used for a variety of purposes, such as analysing flight performance, identifying areas for improvement, and detecting

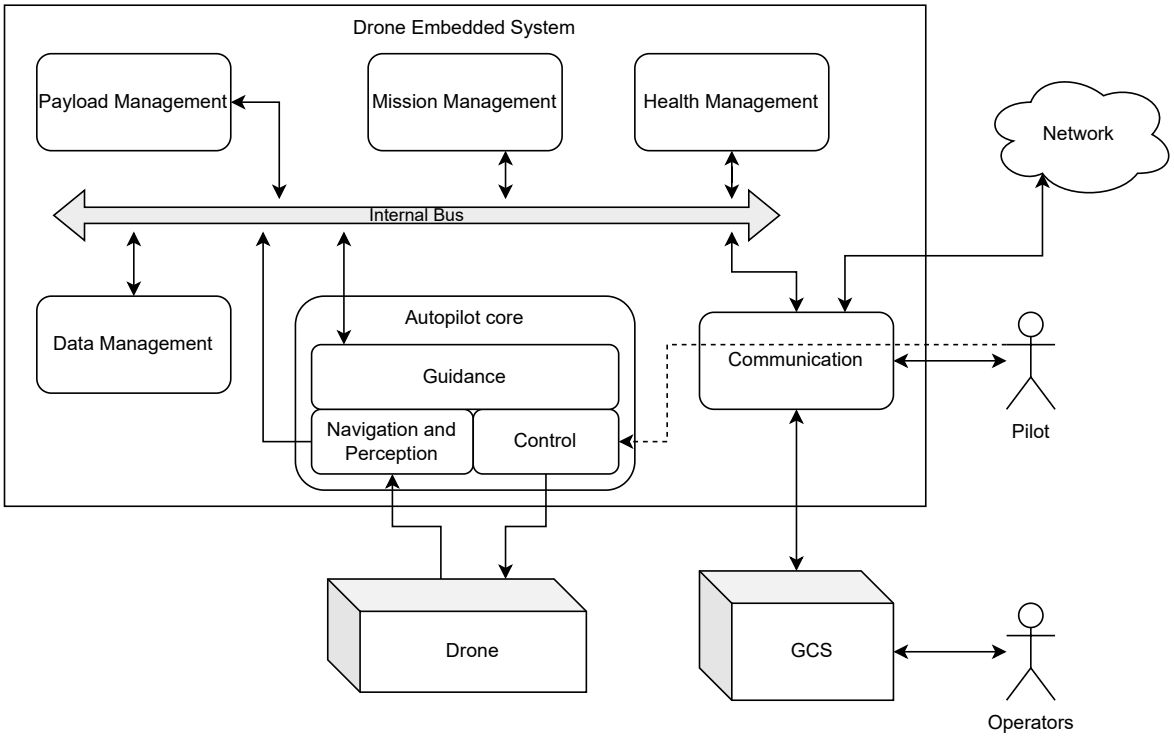

Figure 7.6: Proposed functional reference architecture

anomalies or failures in the drone's subsystems, optimizing data storage and ensuring that it is accurate and reliable. Additionally, the module may provide real-time telemetry data to drone operators, enabling them to monitor the drone's performance during flight.

The Communication module is responsible for establishing and maintaining communication links between the drone and its operator or other external systems. Additionally, it may include advanced encryption and authentication mechanisms to ensure the security and privacy of the communication links. It often allows an override channel such that a pilot can take over the drone's control (represented in the figure as a dashed arrow).

7.4 General Architecture Implementation

While the aforementioned concepts remain in an abstract level, serving as guidelines to using the proposed architecture, we can also propose its implementation. Nevertheless, instead of simply creating a new DSML, we provide the concepts to either create a new DSML or extending existing ones, so they can contemplate the desired characteristics. As existing bases to be extended, we consider AADL and MARTE.

Functions: A function is a programmatic 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. In AADL, it is represented by the concept of a subprogram, while in MARTE it is represented by a step or a UML operation.

Function inputs and outputs: A function is characterized by a set of typed inputs and typed outputs. In AADL, it is represented by the concept of subprogram parameters, while in MARTE it is represented by UML parameters.

Same-thread communication: Within the same thread, communication 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 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. AADL represents them by parameters connections, while MARTE uses object flows in UML activity diagrams.

Function hierarquical decomposition: Functions can often 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 complex systems. Nevertheless, it is not mandatory for a language addressing a method where an existing system is represented. AADL does not provide such a concept, and MARTE might promote hierarchical decomposition with the use of UML components and sub-components.

Function logics: Depending on the needs addressed by the target language, the language may or may not allow expressing the function logic. Furthermore, some formalisms are more adequate to express each logic. For example, a 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). Function logics are interesting to be expressed in the goal of checking functional properties. AADL provides the concept of state machines and modes, and MARTE relies on the capabilities of UML for such (state machines, activity diagrams and other behavioural representations).

Threads: A function must be executed in a thread of execution. 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. AADL contains threads, and MARTE provides concepts such as swSchedulableResource, SaScenario, WorkloadEvent, etc.

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. Various schedulers exist for single-core processors, having statically defined priorities (e.g., Rate Monotonic) or a dynamically defined behaviour (e.g., First In First Out). On a multicore system, scheduling is more complex. AADL proposes concepts such as DispatchProtocol, while MARTE provides

concepts such as computingResource.

Inter-thread communication: Most multi-thread OSs offer several types of inter-thread communications. One of the most powerful set of communications is provided by the POSIX 1003.1 pthread standard, and includes 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 become impossible [Bur99].

Thread-level static scheduler: In the analysed autopilots, within their monolithic implementations, 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 a static scheduling table and 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 algorithms based on rules-of-thumb when customizing the autopilot. 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. This concept cannot be natively represented in AADL nor MARTE.

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. Usually, OSs implement processes by using virtual memory addressing, which requires a specific hardware interface provided by a Memory Management Unit (MMU). The fact that most process decomposition is based on virtual memory implies that it is impossible for a variable in a process memory to be addressed by another process. As a result, inter-process communication requires specific use of functions provided by the OS. In AADL, these are represented by processes and related concepts, while in MARTE they are represented by MemoryPartitions and related concepts.

Partitions: A partition is a static memory allocation, as well as time slot allocation, able to

host an OS. It offers the hosted system a communication API as well as an API to access 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. AADL considers partitions within the ARINC 653 property set, and MARTE does it with the ARINC 653 library.

Platform: The platform is the hardware in which the whole mentioned software executes. It will determine how fast calculations are executed, 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 Architecture Description Language (ADL) models should be able to model this platform directly. In AADL, they are represented by processors, memories, buses and devices, while in MARTE they are represented by hwProcessor, hwMemory, hwBus and hwDevice.

Software Bus: Software buses, whether standard or de-facto standard (ad-hoc) buses, are very common in autopilots. 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 function 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, but in many cases this would be a very heavy modelling effort (e.g., for a multi-platform software bus, a possible route for the transported data could be; function to thread to process to OS to platform bus to target platform to target process to target multi-partition OS to hosted OS to target process to target thread and, finally, 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 data with a certain 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, we would be able to rebuild, from the model, the full route taken by the data. Neither AADL nor MARTE provide such concept.

7.5 Practical Analysis: Paparazzi

Following concepts explored in this chapter, Figure 7.7 has been developed to show an extract of functions which are part of any autopilot nowadays. We can see in a common frame the chain from state estimation to actuation, which will dictate its rhythm to the control.

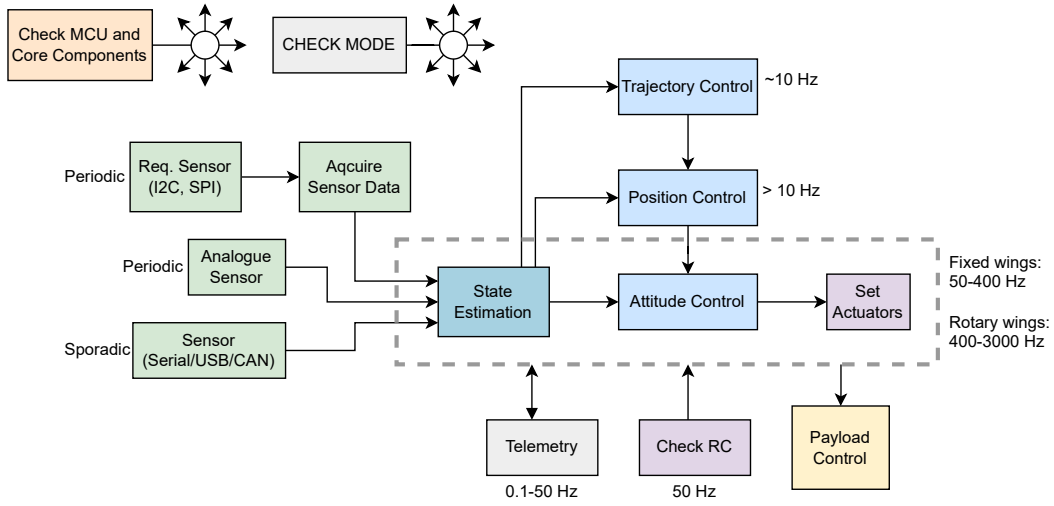

Figure 7.7: Representation of functions in an autopilot

For closely related autopilots, it can be noticed that, depending on the hardware interface used to communicate, three types of constraint can appear. First, if the sensor is analogue, then it is usually read when necessary, at a time chosen by the application (an analogue sensor reading usually takes less than 5 microseconds). Such a sensor can thus be read during a polling phase initiated by the application. Second, if the sensor is connected to a bus (serial, USB, CAN, etc.), then it is usually sending its readings to the autopilot periodically, but using its own clock, which can drift compared to the clock used by the autopilot. We can therefore establish a minimal interval between two successive frames coming from a sensor, but we cannot choose when exactly the frame will be received. As a result, an incoming frame from such a sensor can only be considered sporadic. Third, if the bus used to communicate with the sensor is of type Master/Minion (e.g., I2C, SPI), then a delay has to be considered between the request from the Master to the response from the sensor. Since such a delay can be long (dozens up to hundreds of microseconds), these functions can be decomposed in two parts: (i) the request from the Master, and (ii) handling the response, where a sufficient amount of time has to elapse between the successive executions of the two functions.

Other functional chains expressed in Figure 7.7 have a smaller rate than the core chain state estimation to actuation. For example, the desired rate for position and trajectory control is in the order of magnitude of 10 Hz. Inputs received through legacy Radio Command (RC) is typically 40 Hz, while Datalink received from, and Telemetry transmitted to the ground station have a frequency typically ranging from 0.1 Hz to 50 Hz. The custom payload may require any frequency depending on its nature: from even higher than the core autopilot

frequency, down to less than one hertz.

Note that, on each functional chain, the current state (Fly-by-Wire, Stop, Take off, etc.) shall be the same, otherwise the behaviour of the system may be inconsistent. As a result, a mode change shall occur between two executions of the chain: at the beginning or at the end of its execution.

Nowadays, to guarantee the correct temporal sequence, Paparazzi splits the functions into a predefined list of task groups, shown in Figure 7.8. The selection and ordering of these tasks are the results of an expert-based functional analysis.

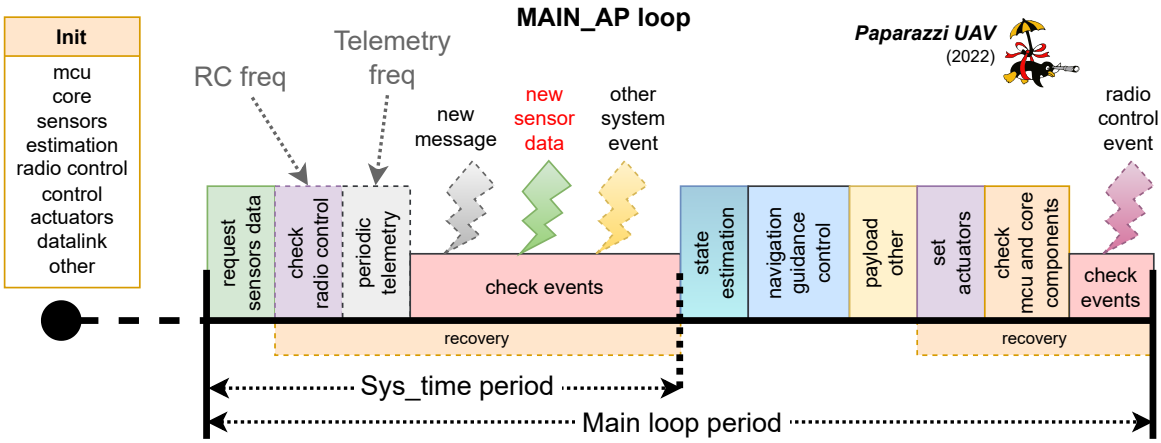

Figure 7.8: Main loop temporal sequence [Hat+22]

After the initialization phase, an infinite loop with a fixed desired frequency is started, related to the *Main loop period* in the figure. Each time the main periodic function is called, the periodic functions from each task group are called in the order presented in the figure, mostly the same as during the initialization phase, but with some modifications and timing constraints. The sensors used for state estimation, when called, initiate a transaction to get new data, which takes time to yield the acquired information and may present jitter. State estimation filters and the control stack should be executed only after sensor data are available, so they are shifted of *Sys_time period* relative to the start of data requisition. The design choice in Paparazzi is to arbitrarily set *Sys_time period* to half of the main loop period and alternate sensor reading and control on this base frequency.

Scheduling the autopilot functions in this scenario requires analysing some characteristics of the functions to be scheduled. Once the dependency has been resolved and the internal order has been defined, assuming a main loop frequency of 1kHz, functions with a period of 1ms have to be scheduled during each iteration. However, functions that have a period multiple of 1ms have a degree of freedom relative to the choice of which loops will host their execution. For example, neglecting any precedence constraints, two 500 Hz functions can be executed in alternating loops such that they will never be executed in the same loop. This strategy can better distribute the workload, helping to avoid the main loop taking more than its defined period to execute.

The choice of the loop where to execute each function is not trivial, as it is equivalent to choosing an offset for periodic functions, which has been explored in chapter 3. Even though some efficient heuristic methods have already been proposed to deal with the problem for independent functions, they cannot be trivially adapted to functions subject to precedence constraints.

Part III

Conclusion

Conclusions and Perspectives

In this thesis, part of the COMP4DRONES project, the context of drone development is assessed, focused on autonomous drones in Europe. After taking drone technology history into account, as well as its current status and deployment, we realize drones are nowadays potentially able to perform multiple kinds of tasks, sparing humans of tedious or dangerous activities. Yet, their use is still limited to very specific domains, their adaptation to specific tasks is time-consuming and error-prone, and their embedded systems cannot be easily validated with respect to potential critical failures. Therefore, their use becomes expensive and risky. In addition, current legislation regarding drone design and operation does not concern embedded software aspects, considering it a black-box and ignoring that autopilot bugs might trigger dangerous behaviours.

It becomes clear that a deeper understanding of drone autopilots is essential for bringing their full potential to life. Shedding light into drone embedded systems is fundamental to evaluating its design and preventing malfunctions, either by standard design practices or by specialized analysis tools. Hence, this thesis fills part of this knowledge gap in the hope of aiding the development of key technological frameworks for drones, on top of which drone technology in Europe can thrive and, consequently, improve the quality of life of our society.

At first, this thesis analyses drone autopilots as real-time embedded systems so that safety aspects can be highlighted. A modular structure could be identified in the most significant open-source autopilots, and the monolithic cyclic executive strategy is identified as the preferred approach to implementing the control algorithms. A software bus (middleware) is the technology that provides the necessary modularity to the software, becoming responsible for guaranteeing the safe transmission of data between different modules. Also, scheduling the cyclic executive has been shown to be a critical aspect to guarantee drone safety, and currently used strategies have room for improvement.

As cyclic executives can be seen at their basis as offset-free systems, we developed the GCD+ algorithm to provide an offset-free system with an offset assignment that avoids critical instants. In a cyclic executive, avoiding critical instants means avoiding, in a single execution cycle, that several modules be put in the execution queue, which potentially results in an execution time that exceeds the time-limit provided for the specific cycle. The algorithm can also be used in a communication link (for example, a telemetry buffer) as to avoid several messages being put in the buffer at the same time, which could result in the loss of messages due to memory limitations.

GCD+ was shown to outperform methods in the state of the art, under the condition that the offset-free system has a sufficiently large GCD of the entities' periods (entities can be threads, execution modules or messages). Its ability to break a very complex problem into smaller subproblems explains how it can provide much better results than the state of the art, even though its choices for subproblem solving can be further improved. However, if the entities in the system are not independent (for example, module A needs to wait a given

amount of time to be executed after module B is executed), GCD+ cannot be used as-is, and an adaptation needs to be made.

Later, GCD+ was adapted to a simplified model of switched networks, where guaranteeing low latency can significantly improve the behaviour of the network. The adaptation of GCD+ is called GCD#, and to the best of our knowledge, it is the only method available in the literature that does not require an analysis of the whole hyperperiod of the system in order to propose initial offsets. While the methods that are mostly used in the State of the Art may not be able to find a solution that satisfies every input constraint, and when they do, it may take a significant amount of time to finish their calculations, GCD# presents itself as a more scalable option, which always proposes a solution. GCD# can be adapted to a more realistic model of switched networks, accounting for better represented technological delays, time-based schedules for each switch in the network, and for hardware limitations such as the number of queues. Furthermore, other arrival patterns can be considered in addition to strictly periodic.

Once real-time aspects have been assessed, the adaptability of drone autopilot architecture is evaluated, given that customization is an important step in deploying useful drones. For such, the most relevant architectures and standards in the robotics and aeronautics domain have been studied. Layered and modular architectures have been shown to be important aspects to guarantee the desired modularity in drone systems, and MDE is presented as a powerful tool to consider these aspects.

One of the typical approaches used to customize drones for one's specific application is to use their autopilot as a Component Off-The-Shelf (COTS), ready to use – a black-box. For such an approach, a modelling language based on ROS was developed: RoBMEX. This language's purpose is to instantiate ROS systems that can be deployed as GCSs or companion-boards. For that, RoBMEX relies on three modelling layers: 1) modelling ROS systems; 2) modelling general simple algorithms; and 3) modelling missions as behaviour trees. RoBMEX can be upgraded to generating ROS2 systems, and have its framework aligned to allow interoperability with real-time analysis tools. In addition, this framework can be extended to swarm missions, and have an improved graphical instantiation interface.

Another typical approach for the customization of drones is to use their autopilots as Modifiable Off-The-Shelf (MOTS) products: although they are ready to use at their acquisition, their code is later customized for the user's specific needs. However, autopilot customization might not be straightforward. In order to ease such task and promote interoperability between modules of different providers, a reference architecture was proposed based on current architectures.

The modular architecture previously identified in the most relevant open-source autopilots has a very clear set of roles, which need to be performed by execution modules in order for the drone to present the desired behaviour. These roles have been highlighted and organized as a reference architecture, bringing out the importance of the middleware as a software bus, and accentuating the relation between some roles and the degree of autonomy of a drone. Additionally, a list of the most fundamental concepts required for any drone autopilot

standard (regarding modularity and safety) has been assembled to guide any efforts to create new architecture implementations or adapt already existing ones, such as AADL or MARTE.

With the results obtained in this thesis, we hope to contribute to create an easier pathway to unite two fundamental but opposing concepts in autonomous drone development: modularity and safety. It is clear that both aspects are crucial for allowing drones to achieve their full potential, but with the presented tools and architectures, their opposition can be mitigated to allow more customizable, yet safer, drone designs. Future works might contribute to reduce the gap between low-level requirements (such as those presented in this thesis) and high-level requirements that may come from risk-analysis approaches and regulatory entities.

Part IV

Appendices

Appendix A

3SAT to Simultaneous Incongruences

The Simultaneous Incongruences (SI) problem is NP-complete [GJ79, problem AN2]. The authors state that SI can be obtained by a transformation from the well-known NP-complete problem, 3SAT. However, that transformation is not clear, to the best of our knowledge, anywhere in the scientific literature. Therefore, based on an Internet forum question regarding the same subject [FT17], we developed the following reasoning.

An instance of a 3SAT problem with m expressions can be represented by Equation A.1.

$$\bigwedge_{i=1}^{m} \left(f_{i_1}(v_{i_1}) \vee f_{i_2}(v_{i_2}) \vee f_{i_3}(v_{i_3})\right)_i \tag{A.1}$$

It translates to a set of logical expressions composed of three single-variable expressions united by a logical disjunctions (OR operator, represented by $\vee$), the set itself united by logical conjunctions (AND operator, represented by $\wedge$). v_{i_1}, v_{i_2} and v_{i_3} are three distinct boolean variables that belong to a finite set of n boolean variables, and $f(v) = v$ or $f(v) = \neg v$, depending on the expression i and the corresponding boolean variable. The problem is: is there any boolean variable affectation (i.e., a defined TRUE or FALSE value for each variable v) that makes the given conjunction of disjunctions assume a TRUE value?

Every instance of 3SAT can be transformed to an instance of SI by a set of polynomial operations (after a single pseudo-polynomial definition of prime numbers is made to encode variables). In order to prove it, let us first represent an instance of SI:

> Given a set of n ordered pairs $\{(a_i, b_i) \in \mathbb{N}^2 \mid a_i < b_i \ \forall i\}$, is there an integer value X such that the following equation is true for all $i \in [1, n]$?
>
> $$X \not\equiv a_i \mod b_i \tag{A.2}$$

From an instance of 3SAT of n variables, each variable v_k can be related to a specific integer in a set of co-prime numbers (for example, the first n primes p_k) so that the valuation

of that variable is coded as follows:

$$X \mod p_k \equiv \begin{cases} 0 & \text{, if } v_k = \text{FALSE} \\ 1 & \text{, if } v_k = \text{TRUE} \end{cases} \tag{A.3}$$

Reciprocally:

$$v_k = \begin{cases} \text{FALSE} & \text{, if } X \equiv 0 \mod p_k \\ \text{TRUE} & \text{, if } X \equiv 1 \mod p_k \end{cases} \tag{A.4}$$

In order for this coding to work, the following constraints must be added to the final SI problem for each p_k used:

$$X \not\equiv y \mod p_k \; \forall y \mid 2 \leq y < p_k \tag{A.5}$$

Every triplet in the 3SAT is of the form $f_{i_1}(v_{i_1}) \vee f_{i_2}(v_{i_2}) \vee f_{i_3}(v_{i_3})$, where at least one of the functions must output the value TRUE so that the triplet also outputs TRUE. Since triplets are united by conjunctions between each other, any triplet with a FALSE output will make the valuation of the whole 3SAT expression turn out to be FALSE. For each triplet, an equivalent expression can be found in SI meaning that each triplet must output a TRUE value.

Each triplet will be associated with an integer M_i. If $f_{i_1}(v_{i_1}) = v_{i_1}$, we must find for this triplet, using the extended Euclidean algorithm and Bézout's Identity, a certain number M_i such that $M_i \equiv 0 \mod p_{i_1}$. Otherwise, if $f_{i_1}(v_{i_1}) = \neg v_{i_1}$, we must find M_i such that $M_i \equiv 1 \mod p_{i_1}$. Doing the same reasoning for v_{i_2} and v_{i_3}, we try to find the same M_i such that the three congruences are respected. Then, we can add the following incongruence to the SI problem:

$$X \not\equiv M_i \mod (p_{i_1} \cdot p_{i_2} \cdot p_{i_3}) \tag{A.6}$$

Then, from the SI problem, $a_i = M_i$ and $b_i = p_{i_1} \cdot p_{i_2} \cdot p_{i_3}$.

To better understand this coding, let us do the reversed path. In order to have a valuation of the 3SAT instance that does not work, the only way that we might choose a number X such that its translation into the variables of 3SAT makes the triplet i output FALSE is if the incongruence is not respected, i.e., if we chose X' such that $X' \equiv M_i \mod (p_{i_1} \cdot p_{i_2} \cdot p_{i_3})$.

This is because, if $X' \equiv M_i \mod (p_{i_1} \cdot p_{i_2} \cdot p_{i_3})$, then $X' \equiv M_i \mod p_{i_j} \; \forall j \in \{1, 2, 3\}$, and we chose M_i such that $M_i \equiv 0 \mod p_{i_j}$ if $f_{i_j} = v_{i_j}$, and 1 otherwise. Knowing the translation from X to v is done using Equation A.4, we see that f_{i_j} must output FALSE $\forall j \in \{1, 2, 3\}$:

- if $M_i \equiv 0 \mod p_{i_j}$, then $X \equiv 0 \mod p_{i_j}$ and $v_{i_j} = \text{FALSE}$, but from the choice of M_i, $f_{i_j} = v_{i_j} = \text{FALSE}$;
- if $M_i \equiv 1 \mod p_{i_j}$, then $X \equiv 1 \mod p_{i_j}$ and $v_{i_j} = \text{TRUE}$, but from the choice of M_i, $f_{i_j} = \neg v_{i_j} = \text{FALSE}$.

Example: Suppose we have an instance of 3SAT in which there is a triplet as follows:

$$v_1 \vee v_{13} \vee \neg v_{25} \tag{A.7}$$

The relative primes can be (from a pseudo-polynomial search made before the transformation) the first, thirteenth and twenty-fifth primes, i.e., 2, 41 and 97.

$$\begin{aligned} p_1 &= 2 \\ p_{13} &= 41 \\ p_{25} &= 97 \end{aligned} \tag{A.8}$$

We then must add the following sets of incongruences to the final SI problem so the encoding is valid:

$$\begin{aligned} X &\not\equiv y \mod 41 \ \forall y \mid 2 \leq y < 41 \\ X &\not\equiv y \mod 97 \ \forall y \mid 2 \leq y < 97 \end{aligned} \tag{A.9}$$

And we find M such that:

$$\begin{aligned} M &\equiv 0 \mod 2 \\ M &\equiv 0 \mod 41 \\ M &\equiv 1 \mod 97 \end{aligned} \tag{A.10}$$

By construction (Bézout's coefficients, found in polynomial time, and Bézout's identity), the set of congruences is solved in polynomial time, and we have (knowing that $2 \cdot 41 \cdot 97 = 7954$):

$$M \equiv 6888 \mod 7954 \tag{A.11}$$

So, we add to the final SI the incongruence:

$$X \not\equiv 6888 \mod 7954 \tag{A.12}$$

Therefore, the set below, put under SI, is equivalent to the 3SAT triplet aforementioned, and a solution to this SI problem can be transformed in polynomial time into a solution for

the 3SAT instance:

$$\{(a_i, b_i)\} = \left\{ \begin{array}{c} (2, 41) \\ (3, 41) \\ (4, 41) \\ \vdots \\ (39, 41) \\ (40, 41) \\ (2, 97) \\ (3, 97) \\ (4, 97) \\ \vdots \\ (95, 97) \\ (96, 97) \\ (6888, 7954) \end{array} \right\} \tag{A.13}$$

Note that, even though several conditions might need to be created for each 3SAT variable (Equation A.9), the transformation is still polynomial. Each 3SAT variable v_k creates $p_k - 2$ incongruences for the encoding to work, and then each triplet in the 3SAT problem generates one incongruence. The amount of incongruences generated only by the encoding of n 3SAT variables is given by the following expression.

$$\sum_{i=1}^{n} (p_k - 2) \tag{A.14}$$

Knowing that $p_k < k \cdot (\ln k + \ln \ln k + 2)$ for $k \geq 6$ [Ros41, theorems 28 and 29], we can easily see that $p_k < 2k^2$. Considering that $\ln x < x$ and $\ln \ln x < \ln x$, then $p_k < k \cdot (k + k + 2)$ for all $k \geq 6$, and by testing smaller values of k we see that this inequality is also true for $1 \leq k \leq 5$.

Therefore, the sum in Equation A.14 must be smaller than twice the sum of the squares of the first n natural numbers. The sum of the first n squares is $n \cdot (n + 1) \cdot (2n + 1)/6$, hence $\mathcal{O}(n^3)$. We can then conclude that the amount of incongruences generated by the encoding is bound by a polynomial function with respect to the amount of variables n.

Bibliography

Bibliography

[Abb+20] Messaoud Abbas et al. “Formal modeling and verification of UML Activity Diagrams (UAD) with FoCaLiZe”. In: *Journal of Systems Architecture* (2020), p. 101911 (cit. on p. 115).

[Ale18] Kozera Cyprian Aleksander. “Military use of unmanned aerial vehicles–a historical study”. In: *Safety & Defense* 4 (2018), pp. 17–21 (cit. on pp. 3, 6).

[AR92] Radio Technical Commission for Aeronautics (RTCA). *Software considerations in airborne systems and equipment certification*. SC 167. RTCA, Incorporated, 1992 (cit. on p. 20).

[Ari] *ARINC 664 Part 7: Avionics Full Duplex Switched Ethernet (AFDX)*. Tech. rep. INC Aeronautical Radio, 2005 (cit. on p. 69).

[ASN16] Sebastian Altmeyer, Sakthivel Manikandan Sundharam, and Nicolas Navet. “The case for FIFO real-time scheduling”. In: (2016) (cit. on pp. 37, 39, 55).

[Ass+04] MISRA The Motor Industry Software Reliability Association et al. *Guidelines for the Use of the C Language in Critical Systems*. 2004 (cit. on p. 21).

[Aut] Auterion. *The story of PX4 and Pixhawk*. `https://auterion.com/company/the-history-of-pixhawk/`. Accessed: 2022-08-08 (cit. on p. 13).

[BBH18] Roberto Bagnara, Abramo Bagnara, and Patricia M Hill. “The misra c coding standard and its role in the development and analysis of safety-and security-critical embedded software”. In: *International Static Analysis Symposium*. Springer. 2018, pp. 5–23 (cit. on p. 21).

[BC11] Chaitanya Belwal and Albert MK Cheng. “Generating bounded task periods for experimental schedulability analysis”. In: *2011 IFIP 9th International Conference on Embedded and Ubiquitous Computing*. IEEE. 2011, pp. 249–254 (cit. on p. 48).

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. “Model-driven software engineering in practice”. In: *Synthesis lectures on software engineering* 3.1 (2017), pp. 1–207 (cit. on p. 85).

[BH08] Pascal Brisset and Gautier Hattenberger. “Multi-UAV control with the paparazzi system”. In: *HUMOUS 2008, conference on humans operating unmanned systems*. Citeseer. 2008, pp–xxxx (cit. on pp. 14, 39).

[Boz+19] Darko Bozhinoski et al. “Managing safety and mission completion via collective run-time adaptation”. In: *Journal of systems architecture* 95 (2019), pp. 19–35 (cit. on p. 115).

[BRI14] BRICS. *bride - ROS Wiki*. `http://wiki.ros.org/bride`. [Online; accessed June 22, 2020]. 2014 (cit. on pp. 90, 114).

[Bro+11] Vicent Brocal et al. “Task period selection to minimize hyperperiod”. In: *ETFA2011*. IEEE. 2011, pp. 1–4 (cit. on p. 48).

[Bru15] Davide Brugali. “Model-driven software engineering in robotics: Models are designed to use the relevant things, thereby reducing the complexity and cost in the field of robotics”. In: *IEEE Robotics & Automation Magazine* 22.3 (2015), pp. 155–166 (cit. on pp. 90, 114).

[Bur99] Alan Burns. “The ravenscar profile”. In: *ACM SIGAda Ada Letters* 19.4 (1999), pp. 49–52 (cit. on p. 128).

[Cas+19] Daniel Casini et al. “Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling”. In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019 (cit. on p. 115).

[CGG04] Annie Choquet-Geniet and Emmanuel Grolleau. “Minimal schedulability interval for real-time systems of periodic tasks with offsets”. In: *Theoretical computer science* 310.1-3 (2004), pp. 117–134 (cit. on p. 56).

[COA17] Silviu S Craciunas, R Serna Oliver, and T Ag. “An overview of scheduling mechanisms for time-sensitive networks”. In: *Proceedings of the Real-time summer school LÉcole dÉté Temps Réel (ETR)* (2017), pp. 1551–3203 (cit. on p. 43).

[Cot+06] Carle Cote et al. “Robotic software integration using MARIE”. In: *International Journal of Advanced Robotic Systems* 3.1 (2006), p. 10 (cit. on p. 90).

[Den+16] Stampfer Dennis et al. “The SmartMDSD toolchain: An integrated MDSD workflow and integrated development environment (IDE) for robotics software”. In: *Journal of Software Engineering for Robotics* (2016) (cit. on p. 90).

[Dho+12] Saadia Dhouib et al. “Robotml, a domain-specific language to design, simulate and deploy robotic applications”. In: *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer. 2012, pp. 149–160 (cit. on pp. 90, 114).

[Doc] *PX4 User Guide*. `https://docs.px4.io/main/en/`. Accessed: 2022-08-08. 2022 (cit. on pp. 14, 15).

[Dro15] DroneKit. *DroneKit*. `https://dronekit.io/`. [Online; accessed 2nd February 2020]. 2015 (cit. on p. 89).

[Dro19] Dronecode. *Introduction · MAVLink Developer Guide*. `https://mavlink.io/`. [Online; accessed 16th December 2019]. 2019 (cit. on p. 90).

[DZ83] John D Day and Hubert Zimmermann. “The OSI reference model”. In: *Proceedings of the IEEE* 71.12 (1983), pp. 1334–1340 (cit. on p. 41).

[Erm18] Vladimir Ermakov. *mavros - ROS wiki*. `http://wiki.ros.org/mavros`. [Online; accessed January 14, 2020]. 2018 (cit. on pp. 91, 107).

[Eti18] Daniel Etiemble. “45-year CPU evolution: one law and two equations”. In: *arXiv preprint arXiv:1803.00254* (2018) (cit. on p. 3).

[EU17] RobMoSys EU. *Project (2017–2020). RobMoSys: Composable Models and Software for Robtics Systems-Towards an EU Digital Industrial Platform for Robotics*. 2017 (cit. on p. 94).

[Fau+07] Madeleine Faugere et al. “Marte: Also an UML profile for modeling AADL applications”. In: *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*. IEEE. 2007, pp. 359–364 (cit. on pp. 88, 89).

[Fei+04] Peter H Feiler et al. “An overview of the SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering”. In: *IFIP World Computer Congress, TC 2*. Springer. 2004, pp. 3–15 (cit. on p. 87).

[Fei19] Peter Feiler. *The Open Source AADL Tool Environment (OSATE)*. Tech. rep. Carnegie Mellon University Software Engineering Institute, 2019 (cit. on p. 88).

[FH20] Mohammed Foughali and Pierre-Emmanuel Hladik. “Bridging the gap between formal verification and schedulability analysis: The case of robotics”. In: *Journal of Systems Architecture* 111 (2020), p. 101817 (cit. on p. 115).

[FLY19] FLYAQ. *FLYAQ*. `http://www.flyaq.it/`. [Online; accessed 10th January 2020]. 2019 (cit. on pp. 89, 114).

[Fly19] Inc. FlytBase. *FlytOS: Operating System for Drones*. `https://flytbase.com/flytos/`. [Online; accessed 10th January 2020]. 2019 (cit. on p. 89).

[FT17] Yuval Filmus and TheoryQuest1. *Computer Science Stack Exchange: Transforming 3 SAT to simultaneous in-congruence problem?* `https://cs.stackexchange.com/questions/72562/transforming-3-sat-to-simultaneous-in-congruence-problem`. [Online; accessed March 3rd, 2023]. 2017 (cit. on p. 141).

[Für+09] Simon Fürst et al. “AUTOSAR–A Worldwide Standard is on the Road”. In: *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*. Vol. 62. 2009, p. 5 (cit. on p. 19).

[GBD20] David Griffin, Iain Bate, and Robert I Davis. “Generating utilization vectors for the systematic evaluation of schedulability tests”. In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2020, pp. 76–88 (cit. on p. 55).

[GD97] Joël Goossens and Raymond Devillers. “The non-optimality of the monotonic priority assignments for hard real-time offset free systems”. In: *Real-Time Systems* 13 (1997), pp. 107–126 (cit. on p. 37).

[GGN06] Mathieu Grenier, Joël Goossens, and Nicolas Navet. “Near-optimal fixed priority preemptive scheduling of offset free systems”. In: *14th International Conference on Real-Time and Networks Systems (RTNS’06)*. 2006, pp. 35–42 (cit. on p. 39).

[GHN08] Mathieu Grenier, Lionel Havet, and Nicolas Navet. “Pushing the Limits of CAN - Scheduling Frames with Offsets Provides a Major Performance Boost”. In: *4th European Congress on Embedded Real Time Software (ERTS 2008)* (2008) (cit. on pp. 39, 56, 57).

[GJ79] Michael R Garey and David S Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Vol. 174. W. H. Freeman and Company, 1979 (cit. on pp. 47, 141).

[GM01] Joel Goossens and Christophe Macq. “Limitation of the hyper-period in real-time periodic task set generation”. In: *In Proceedings of the RTS Embedded System (RTS'01*. Citeseer. 2001 (cit. on p. 55).

[Goo03] Joël Goossens. “Scheduling of offset free systems”. In: *Real-Time Systems* 24.2 (2003), pp. 239–258 (cit. on pp. 39, 56, 57).

[Gro15] Object Management Group. *Data Distribution Service (DDS) Version 1.4*. Tech. rep. formal/15-04-10. OMG, Mar. 2015 (cit. on pp. 42, 92).

[Gro17] Teal Group. “2017 world civil unmanned aerial systems market profile & forecast”. In: (2017) (cit. on p. 4).

[Gro19a] Object Management Group. *The Real-time Publish-Subscribe Protocol DDS Interoperability Wire Protocol (DDSI-RTPS) Specification, Version 2.3*. Tech. rep. formal/19-04-03. OMG, May 2019 (cit. on pp. 42, 92).

[Gro19b] Teal Group. “2019 world civil unmanned aerial systems market profile & forecast”. In: (2019) (cit. on p. 4).

[Gro99] Emmanuel Grolleau. “Ordonnancement temps réel hors-ligne optimal à l'aide de réseaux de pétri en environnement monoprocesseur et multiprocesseur”. PhD thesis. Poitiers, 1999 (cit. on p. 38).

[Hat+22] Gautier Hattenberger et al. “Micro-drone autopilot architecture for efficient static scheduling”. In: *13th International Micro Air Vehicle Conference*. Delft, The Netherlands, Sept. 2022 (cit. on pp. 39, 56, 131).

[HBG14] Gautier Hattenberger, Murat Bronz, and Michel Gorraz. “Using the paparazzi UAV system for scientific research”. In: *IMAV 2014, International Micro Air Vehicle Conference and Competition 2014*. 2014, pp–247 (cit. on p. 39).

[Iso] *Road vehicles — Functional safety*. Standard. Geneva, CH: International Organization for Standardization, Dec. 2018 (cit. on p. 18).

[Jou+08] Frédéric Jouault et al. “ATL: A model transformation tool”. In: *Science of computer programming* 72.1-2 (2008), pp. 31–39 (cit. on p. 86).

[JSM91] Kevin Jeffay, Donald F Stanat, and Charles U Martel. “On non-preemptive scheduling of periodic and sporadic tasks”. In: *IEEE real-time systems symposium*. Citeseer. 1991, pp. 129–139 (cit. on p. 38).

[JU19] ECSEL JU. *WHAT IS COMP4DRONES?* 2019. URL: `https://www.comp4drones.eu/` (visited on 06/27/2022) (cit. on p. 22).

[Ken12a] Farid Kendoul. “Survey of advances in guidance, navigation, and control of unmanned rotorcraft systems”. In: *Journal of Field Robotics* 29 (2 Mar. 2012), pp. 315–378 (cit. on pp. 8, 9).

[Ken12b] Farid Kendoul. “Survey of advances in guidance, navigation, and control of unmanned rotorcraft systems”. In: *Journal of Field Robotics* 29.2 (2012), pp. 315–378 (cit. on pp. 123, 125).

[Koo14] Phil Koopman. “A case study of Toyota unintended acceleration and software safety”. In: *Presentation. Sept* (2014) (cit. on p. 17).

[Koo18] Philip Koopman. “Practical experience report: Automotive safety practices vs. accepted principles”. In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2018, pp. 3–11 (cit. on p. 17).

[KS08] Omar Kermia and Yves Sorel. “Schedulability analysis for non-preemptive tasks under strict periodicity constraints”. In: *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2008, pp. 25–32 (cit. on p. 46).

[Lad+22] Matheus Ladeira et al. “Scheduling Offset-Free Systems Under FIFO Priority Protocol”. In: *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Vol. 231. 2022 (cit. on p. 48).

[LIA21] LIAS. *Documentation - RoBMEX - Welcome to the LIAS Forge*. `https://forge.lias-lab.fr/projects/robmex/wiki/Documentation`. [Online; accessed February 4th, 2021]. 2021 (cit. on p. 115).

[LOG21] Matheus Ladeira, Yassine Ouhammou, and Emmanuel Grolleau. “RoBMEX: ROS-based modelling framework for end-users and experts”. In: *Journal of Systems Architecture* 117 (2021), p. 102089 (cit. on p. 97).

[Lot+17] Alex Lotz et al. “Towards a stepwise variability management process for complex systems: A robotics perspective”. In: *Artificial Intelligence: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2017, pp. 2411–2430 (cit. on pp. 90, 114).

[Lui+04] Sha Lui et al. “Real Time Scheduling Theory: A Historical Perspective”. In: 28 (2004) (cit. on pp. 29, 36).

[MB08] Leonardo de Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340 (cit. on p. 44).

[MB20] Rico Merkert and James Bushell. “Managing the drone revolution: A systematic literature review into the current use of airborne drones and future strategic directions for their effective control”. In: *Journal of Air Transport Management* 89 (2020), p. 101929 (cit. on p. 3).

[MCF03] Stephen J Mellor, Tony Clark, and Takao Futagami. “Model-driven development: guest editors’ introduction. IEEE Software, 20 (5). pp. 14-18. ISSN 0740-7459”. In: *IEEE software* 20.5 (2003), pp. 14–18 (cit. on p. 85).

[MGS12] Mohamed Marouf, Laurent George, and Yves Sorel. “Schedulability analysis for a combination of non-preemptive strict periodic tasks and preemptive sporadic tasks”. In: *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. IEEE. 2012, pp. 1–8 (cit. on p. 46).

[Miy+19] Alvaro Miyazawa et al. “RoboChart: modelling and verification of the functional behaviour of robotic applications”. In: *Software & Systems Modeling* 18.5 (2019), pp. 3097–3149 (cit. on pp. 89, 114).

[MKK16] Panos Marantos, Yannis Koveos, and Kostas J. Kyriakopoulos. “UAV State Estimation Using Adaptive Complementary Filters”. In: *IEEE Transactions on Control Systems Technology* 24.4 (2016), pp. 1214–1226 (cit. on p. 8).

[MRT03] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. “Perspectives on standardization in mobile robot programming: The Carnegie Mellon navigation (CARMEN) toolkit”. In: *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*. Vol. 3. IEEE. 2003, pp. 2436–2441 (cit. on p. 90).

[MS11] Mohamed Marouf and Yves Sorel. “Scheduling non-preemptive hard real-time tasks with strict periods”. In: *ETFA2011*. IEEE. 2011, pp. 1–8 (cit. on p. 46).

[NDB18] Mitra Nasri, Robert I Davis, and Björn B Brandenburg. “FIFO with offsets: High schedulability with low overheads”. In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2018, pp. 271–282 (cit. on pp. 40, 47).

[NF15] Mitra Nasri and Gerhard Fohler. “An efficient method for assigning harmonic periods to hard real-time tasks with period ranges”. In: *2015 27th Euromicro Conference on Real-Time Systems*. IEEE. 2015, pp. 149–159 (cit. on p. 48).

[NF16] Mitra Nasri and Gerhard Fohler. “Non-work-conserving non-preemptive scheduling: motivations, challenges, and potential solutions”. In: *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE. 2016, pp. 165–175 (cit. on p. 37).

[Ngu+18] Thanh-Dat Nguyen et al. “Design and analysis of semaphore precedence constraints: A model-based approach for deterministic communications”. In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 231–236 (cit. on pp. 43, 64).

[Nik+03] Ioannis K Nikolos et al. “Evolutionary algorithm based offline/online path planner for UAV navigation”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 33.6 (2003), pp. 898–912 (cit. on p. 10).

[Nou+19] Réda Nouacer et al. “Framework of Key Enabling Technologies for Safe and Autonomous Drones’ Applications”. In: *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE. 2019, pp. 420–427 (cit. on p. 22).

[OSRF19] Inc. Open Source Robotics Foundation. *Design*. `http://design.ros2.org/`. [Online; accessed 15th April 2020]. 2019 (cit. on p. 92).

[Pan+15] Chaoyi Pang et al. “Topological sorts on DAGs”. In: *Information Processing Letters* 115.2 (2015), pp. 298–301 (cit. on p. 65).

[Pap] Paparazzi developers. *Paparazzi offset generation source code*. `https://github.com/paparazzi/paparazzi/blob/master/sw/tools/generators/gen_periodic.ml`. Accessed: 02-02-2022 (cit. on pp. 39, 57).

[Pap18] PaparazziUAV. *Autopilot generation - PaparazziUAV*. `https://wiki.paparazziuav.org/wiki/Autopilot_generation`. [Online; accessed 16th December 2019]. 2018 (cit. on p. 39).

[PC03] Gerardo Pardo-Castellote. “OMG Data-Distribution Service: Architectural Overview”. In: *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE. 2003, pp. 200–206 (cit. on p. 42).

[Per11] Laurent Perron. “Operations Research and Constraint Programming at Google”. In: *Principles and Practice of Constraint Programming – CP 2011.* Ed. by Jimmy Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 2–2 (cit. on p. 44).

[PL05] Rodolfo Pellizzoni and Giuseppe Lipari. “Feasibility analysis of real-time periodic tasks with offsets”. In: *Real-Time Systems* 30.1-2 (2005), pp. 105–128 (cit. on p. 37).

[Pri92] Paul J Prisaznuk. “Integrated modular avionics”. In: *Proceedings of the IEEE 1992 National Aerospace and Electronics Conference@ m_NAECON 1992.* IEEE. 1992, pp. 39–45 (cit. on p. 19).

[RBR12] Ismael Ripoll and Rafael Ballester-Ripoll. “Period selection for minimal hyperperiod in periodic task systems”. In: *IEEE Transactions on Computers* 62.9 (2012), pp. 1813–1822 (cit. on p. 48).

[Rob19a] RobMoSys. *RobMoSys - Composable Models and Software.* `https://robmosys.eu/`. [Online; accessed January 10, 2020]. 2019 (cit. on pp. 90, 114).

[Rob19b] Open Robotics. *ROS.org | About ROS.* `https://www.ros.org/about-ros/`. [Online; accessed January 8, 2020]. 2019 (cit. on p. 91).

[Roq16] Pascal Roques. “MBSE with the ARCADIA Method and the Capella Tool”. In: *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016).* 2016 (cit. on p. 93).

[Ros41] Barkley Rosser. “Explicit bounds for some functions of prime numbers”. In: *American Journal of Mathematics* 63.1 (1941), pp. 211–232 (cit. on p. 144).

[RPC20] Niklas Reusch, Paul Pop, and Silviu S Craciunas. “Work-in-progress: Safe and secure configuration synthesis for TSN using constraint programming”. In: *2020 IEEE Real-Time Systems Symposium (RTSS).* IEEE. 2020, pp. 387–390 (cit. on pp. 67, 81).

[SCM10] S Sieberling, QP Chu, and JA Mulder. “Robust flight control using incremental nonlinear dynamic inversion and angular acceleration prediction”. In: *Journal of guidance, control, and dynamics* 33.6 (2010), pp. 1732–1742 (cit. on p. 10).

[Sel07] Bran Selic. “A systematic approach to domain-specific language design using UML”. In: *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07).* IEEE. 2007, pp. 2–9 (cit. on p. 87).

[Seo+21] Youhwan Seol et al. “Timely survey of time-sensitive networking: Past and future directions”. In: *IEEE Access* 9 (2021), pp. 142506–142527 (cit. on p. 67).

[Ste10] Wilfried Steiner. “An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks”. In: *2010 31st IEEE Real-Time Systems Symposium.* IEEE. 2010, pp. 375–384 (cit. on p. 44).

[Tes98] Nikola Tesla. *Method of and apparatus for controlling mechanism of moving vessels or vehicles*. US613809A. Nov. 1898 (cit. on p. 3).

[Via21] Louis Viard. "Méthodes et outils pour la programmation des systèmes cyber-physiques". PhD thesis. Université de Lorraine, 2021 (cit. on p. 4).

[WB+95] Greg Welch, Gary Bishop, et al. "An introduction to the Kalman filter". In: (1995) (cit. on p. 8).

[Wen+16] Monika Wenger et al. "A model based engineering tool for ROS component compositioning, configuration and generation of deployment information". In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2016, pp. 1–8 (cit. on pp. 90, 114).

[Xu10] Jia Xu. "A method for adjusting the periods of periodic processes to reduce the least common multiple of the period lengths in real-time embedded systems". In: *Proceedings of 2010 IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications*. IEEE. 2010, pp. 288–294 (cit. on p. 48).

[Yom09] Patrick Meumeu Yomsi. "Prise en compte du coùt exact de la préemption dans l'ordonnancement temps réel monoprocesseur avec contraintes multiples". PhD thesis. 2009 (cit. on p. 38).

[YS06] Patrick Meumeu Yomsi and Yves Sorel. "Non-schedulability conditions for off-line scheduling of real-time systems subject to precedence and strict periodicity constraints". In: *2006 IEEE Conference on Emerging Technologies and Factory Automation*. IEEE. 2006, pp. 1–6 (cit. on p. 46).

[YS07] Patrick Meumeu Yomsi and Yves Sorel. "Schedulability analysis using exact number of preemptions and no idle time for real-time systems with precedence and strict periodicity constraints". In: *Proceedings of 15th International Conference on Real-Time and Network Systems, RTNS'07*. 2007 (cit. on p. 46).

[Zha+20] Luxi Zhao et al. "Latency analysis of multiple classes of AVB traffic in TSN with standard credit behavior using network calculus". In: *IEEE Transactions on Industrial Electronics* 68.10 (2020), pp. 10291–10302 (cit. on p. 80).

Résumé étendu

Chapitre 1 - Introduction

Dans le premier chapitre, une introduction générale au sujet des drones est présentée. Des premières versions des drones ont été utilisées dans les deux guerres mondiales, mais l'effort nécessaire pour son utilisation dans le domaine civile était trop élevé. Le développement des composants électroniques en miniature et abordables a réduit les coûts d'implémentation, et ces nouveaux produits disponibles au marché ont permis la création et la production de plusieurs types de drones, qui sont plus proches d'atteindre leurs potentielles applications.

Néanmoins, la complexité de leurs systèmes embarqués peut devenir un obstacle à l'utilisation des drones dans certaines applications, surtout quand il s'agit des drones autonomes. Le logiciel embarqué, en plus de devoir considérer des concepts de plusieurs domaines distincts (hardware, contrôle et stabilité, aérodynamique, etc.), doit aussi être validé au niveau de sa sûreté – une panne du logiciel probablement provoquera une panne totale et, par conséquent, des dégâts matériels ou même des blessures.

Pour gérer la complexité du logiciel embarqué, des solutions open-source (notamment ArduPilot, Paparazzi et PX4) ont pu modulariser le code et séparer les problèmes, de façon que la customisation d'un drone puisse se faire de manière plus simple. Par contre, cette approche peut rendre la validation de sureté plus difficile.

Dans ce contexte, le projet COMP4DRONES réuni 50 organisations européennes avec le but de développer les outils et méthodologies nécessaires pour permettre le développement des drones plus modulaires, mais de façon à garantir leur sûreté de fonctionnement.

La définition de drone autonome (aussi appelé UAV – acronyme en anglais pour véhicule autonome aérien) est précisée dans ce chapitre, aussi bien que celle d'un UAS (système autonome aérien, qui comprend le drone plus la station sol), un autopilote (le logiciel embarqué qui mesure, interprète, planifie et contrôle le mouvement du drone), la navigation (module de calcul de la position, vitesse et accélération du drone), le contrôle (module qui commande les actionneurs et moteurs du drone), le guidage (module de gestion de la trajectoire), l'autonomie (dégrée d'indépendance d'un pilote humain) et le middleware (module logiciel responsable pour l'implémentation d'un bus logiciel dans un système embarqué).

Ensuite, nous décrivons les autopilotes open-source recherchés, en mettant en évidence comment leurs boucles principales se divisent et quelles fonctions sont exécutées. Dans les trois autopilotes mentionnés, les modules de Navigation, Guidage et Contrôle sont bien définis, et la boucle principale est divisée au moins en deux : une boucle rapide et une boucle plus lente. La boucle rapide s'agit du contrôle de l'attitude du drone (la position angulaire et ses dérivées), et la boucle lente s'agit de la position linéaire et ses dérivées. Les différents niveaux d'autonomie proposés par chaque autopilote ont une forte correspondance entre eux,

où l'utilisateur peut choisir si la stabilisation automatique sera faite seulement pour la vitesse angulaire (mode manuel), attitude (mode assisté), position (mode semi-autonome) ou si un concept de mission est présent (mode autonome).

Ensuite, des standards de sûreté actuels sont présentés. Les standards spécifiques aux drones, comme la méthode SORA proposée par JARUS, sont généralistes et ne contemplent pas la problématique du logiciel. Ainsi, nous avons cherché de l'inspiration sur des normes du secteur automotive, comme l'ISO 26262, qui propose la classification des risques par des niveaux ASIL – un ASIL A contient les contraintes les plus faibles, et le niveau ASIL D contient les contraintes les plus fermes. En outre, le standard AUTOSAR propose une interface commune pour faciliter l'intégration des modules des différents fournisseurs.

La sûreté dans le secteur de l'aviation a aussi été évaluée, comme ARINC 653, qui propose l'utilisation des partitions complètement isolées pour l'exécution de différents modules fonctionnels d'un système avionique, et le standard DO-178C, qui définit des niveaux de criticité pour une panne du module logiciel (de DAL E, le moins critique, à DAL A, le plus critique). Les standards MISRA sont aussi une source d'inspiration, pour leurs directives par rapport à l'écriture d'un code source pour un logiciel embarqué.

À la fin du chapitre, le projet COMP4DRONES est décrit : ses cas d'utilisation et ses groupes de travail. La présente thèse fait partie des groupes 3 (Architecture Modulaire Intégrée), 6 (Outils de Conception, Performance et Vérification) et 7 (Exploitation, Vulgarisation, Entraînements et Standardisation).

Chapitre 2 - Contexte en Programmation Temps-Réel

Le deuxième chapitre fournit un contexte en programmation temps réel appliqué aux drones. D'abord, les concepts de base sont définis, comme un processeur (entité programmable), comportement (ensemble d'actions et décisions prises par un algorithme), algorithme (séquence d'instructions élémentaires exécutées par un processeur), programme (entité exécutable, qui peut comporter plusieurs algorithmes), et fonction (sous-division d'un programme).

Ensuite, l'exécutif cyclique est décrit : une boucle infinie qui exécute un ensemble de fonctions de manière séquentielle et ininterrompue. Avec la possibilité d'interrompre une exécution, le concept de fils d'exécution (thread, une séquence d'exécution avec son propre contexte de mémoire) apparait, et avec celui-ci, les concepts de job (une exécution d'un thread), période (intervalle de temps entre deux jobs d'un thread), hyperpériode (intervalle de temps dont lequel un système avec plusieurs threads strictement périodiques répète sa structure d'exécution), système d'exploitation (couche d'abstraction qui fournit un ordonnanceur), procès (isolation logique de mémoire), ordonnanceur (entité qui ordonnance l'exécution des threads), temps de réponse (intervalle entre le déclenchement d'un job et la fin de son exécution), échéance (temps de réponse maximale toléré), temps de réponse au pire cas (WCET – temps de réponse maximale pour un thread dans un certain ordonnanceur), instant critique (configuration des threads dans un ordonnanceur qui amène à la valeur maximale du WCET

d'un thread), etc.

Des stratégies d'ordonnancement de l'état de l'art sont présentées. Elles ont été développées pour essayer de fournir des garanties à propos des temps de réponse des threads par rapport à leurs échéances. Des ordonnanceurs de priorité fixe ont été créés, comme l'ordonnancement à taux monotone (RM) ou l'échéance plus proche en premier (EDF). Ensuite, des ordonnanceurs à priorités dynamiques, comme le premier entré premier sorti (FIFO). Les ordonnanceurs FIFO possèdent des propriétés intéressantes pour les systèmes embarqués critiques, comme le fait que des threads strictement périodiques ordonnancés sous FIFO sont déterministes.

Dans des ordonnanceurs FIFO, souvent les threads peuvent avoir des décalages pour le moment de départ, ce qui peut éviter un instant critique de se produire. Quand le choix du décalage est libre pour un système de threads strictement périodiques, nous parlons d'un système "offset-free".

Pour ces systèmes, le nombre d'affectations de décalages non équivalentes possibles est le résultat d'une explosion combinatoire des périodes des threads, et il est postulé qu'il s'agisse d'un problème de type NP. Face à cette explosion, les techniques pour trouver les décalages adéquats pour un tel système peuvent être de deux types :

1. Des méthodes heuristiques, qui réduisent le temps de calcul des décalages, mais ne garantissent pas que la solution est optimale ;
2. Des méthodes d'optimisation, qui explorent le domaine des solutions possibles pour trouver une solution optimale, mais qui peuvent être sensiblement chronophages.

Les systèmes offset-free ont des propriétés uniques et utiles pour leur prédictibilité, qui sont rappelées par les théorèmes de ce chapitre. Par exemple :

Théorème 2.3.1 : La distance minimale $\Delta_{i,j}$ entre le déclenchement d'un job d'un thread τ_i (avec décalage O_i et période T_i) et d'un déclenchement subséquent d'un job de τ_j (avec décalage O_j et période T_j) est donnée par l'expression $\Delta_{i,j} = (O_j - O_i) \mod \text{PGCD}(T_i, T_j)$.

En dessinant le cercle qui représente les numéros modulo PGCD comme dans la figure 2.8, si nous représentons les instants des déclenchements des threads τ_i et τ_j en plus de leur temps d'exécution, nous pouvons constater si leurs temps d'exécutions se chevauchent. Un chevauchement des temps d'exécution indique la présence d'un potentiel temps d'attente : l'un des threads devra attendre la fin de l'exécution de l'autre, si l'autre thread a atteint son temps d'exécution maximale et que la distance entre les déclenchements est minimale.

Parmi les heuristiques de l'état de l'art, l'heuristique utilisée dans l'autopilote Paparazzi pour ordonnancer des messages dans une fille d'attente de la télémétrie du drone (analogue à un processeur monocœur) est présentée : les décalages sont calculés en fonction d'un pourcentage de la période du message, et ce pourcentage dépend de la cardinalité du message dans la liste exhaustive des messages envoyés par ce lien. Ensuite, l'heuristique proposée par Joël Goossens est mentionnée, qui maximise $\Delta_{i,j}$ pour un sous-ensemble de toutes les

paires de threads du système. Une méthode proposée pour les bus de messages CAN est aussi mentionnée, qui construit l'exécution du système jusqu'à la période la plus grande, et positionne les décalages des messages l'un après l'autre au milieu du plus grand intervalle de temps le moins chargé. D'autres méthodes proposent une adaptation de l'ordonnanceur, mais nous avons choisi de nous concentrer seulement sur le changement de la configuration de ces ordonnanceurs, un changement des données d'entrée plutôt que de leurs algorithmes.

Par ailleurs, quelques stratégies de communication entre différents threads sont explorées, comme l'utilisation des mutexes et des sémaphores. Le cas des threads déployés dans différentes parties physiques est aussi analysé en s'appuyant sur les couches OSI.

1 - Physique : e.g., les bits d'information transmis dans un cable ;
2 - Liaison : adressage physique ;
3 - Réseau : adressage logique ;
4 - Transport : notions de port ;
5 - Session : ouverture et fermeture des sessions ;
6 - Présentation : chiffrement et déchiffrement des données ;
7 - Application : point d'accès aux services du réseau.

Pour abstraire la gestion de ces couches, un middleware peut être utilisé, comme le middleware spécifique à l'autopilote Paparazzi (ABI) ou des middlewares disponibles au marché, comme ROS, ROS2, uORB ou DDS.

La communication entre threads peut provoquer une dépendance de l'exécution par rapport à l'échange de données. Cela produit des relations de précédence entre les threads, où l'exécution d'un job d'un thread doit se passer après l'exécution d'un job du thread précédant. Cette notion peut être appliquée intuitivement quand un système possède des threads de même période, mais elle doit être adaptée pour des threads de périodes différentes, en utilisant par exemple la technique SPC (contrainte de précédence par sémaphore).

Une fois validée la communication entre les threads, des réseaux peuvent être analysés par rapport à ses contraintes temps-réel. Même si des protocoles réseau existent spécifiquement pour les applications temps-réel, une approche très utilisée est l'adoption du protocole Ethernet avec des standards supplémentaires qui lui fournissent une certaine prédictibilité, comme TTEthernet et TSN, qui ordonnancent les messages statiquement.

Pour réaliser cet ordonnancement, une technique répandue est la programmation par contraintes et l'optimisation. La programmation par contraintes crée un modèle mathématique du problème d'ordonnancement et essaye de trouver des paramètres (comme le décalage du temps d'envoi des messages) qui puisse satisfaire toutes les contraintes du problème, comme le fait qu'un seul message peut être envoyé dans un lien de communication dans un instant spécifique. L'optimisation essaye de trouver la solution qui satisfait les contraintes, mais aussi minimise ou maximise une certaine métrique, par exemple, la latence.

Chapitre 3 - GCD+ : Heuristique d'Ordonnancement

Basé sur le contexte d'ordonnancement temps-réel, ce chapitre explore un problème lié à l'ordonnancement de messages dans un lien de communication de la télémétrie de Paparazzi, mais qui a des implications pour l'ordonnancement des threads dans un ordonnanceur FIFO et pour les fonctions dans un exécutif cyclique. Le problème initialement exploré est la perte de messages à laquelle un drone Paparazzi subit pour envoyer des données de télémétrie à une station sol. Ces données se perdent dans une file d'attente due au débordement de la file, qui reçoit trop de messages à la fois dans des moments précis. Une affectation de décalages peut réduire la quantité de messages présents simultanément dans la file d'attente, et une méthode pour la définir est cherchée.

Un utilisateur d'un ordonnanceur des threads peut aussi avoir intérêt à réduire la quantité de threads déclenchés simultanément, vu que cela réduirait la latence maximale et, par conséquent, peut augmenter l'ordonnançabilité d'un système. C'est pourquoi, dans ce chapitre, la modélisation du problème se base sur le concept de thread non préemptible strictement périodique exécuté sur un processeur monocœur, mais qui aura la même modélisation qu'un message strictement périodique dans un lien de communication.

Étant donné un ensemble de threads τ_i définis par leur période T_i et leur temps d'exécution (par exemple, le WCET) C_i, nous voulons trouver un ensemble de décalages O_i de façon à réduire le temps d'attente provoqué par l'exécution non terminée d'autres threads. Un sous-problème de cette question s'agit de trouver le décalage O_{n+1} pour un thread τ_{n+1} qui se rajoute à un système de n threads, chacun déjà avec son propre décalage, et de façon que le nouveau thread ne se déclenche pas en même temps qu'un autre thread. Ce sous-problème est équivalent au problème NP-complet des Incongruences Simultanées (SI), donc nous supposons que le problème de trouver un ensemble de décalages est au moins si difficile que SI. Par conséquent, nous cherchons une méthode heuristique (solution non exacte), vu qu'une solution non exponentielle exacte ne peut exister que si P=NP. Une marge d'amélioration a été identifiée par rapport aux heuristiques de l'état de l'art, et ce chapitre présente une nouvelle heuristique pour trouver des décalages dans un système offset-free : GCD+.

GCD+ est basée sur la construction du cercle qui représente les numéros modulo un certain PGCD. À partir de cette construction, on peut constater rapidement le chevauchement de l'exécution des threads, c'est-à-dire, quand $C_i > \Delta_{i,j}$ ou quand $C_j > \Delta_{j,i}$. Par contre, cette construction ne peut fournir des conclusions valides que pour des paires de threads. GCD+ permet une extrapolation de cette méthode pour que le cercle modulo PGCD puisse héberger plusieurs threads.

En prenant le PGCD de tous les threads, appelé ici Ω, GCD+ définit la taille d'un cycle comme Ω et divise l'exécution du système dans le temps comme une suite infinie de cycles. Les périodes des threads peuvent être définies par un nombre entier de cycles, et ce nombre est appelé sous-période. Basée sur le Théorème du Restant Chinois Généralisé, il est possible de prouver que, pour des sous-périodes qui sont co-premières entre elles, les threads correspondants sont obligés de partager un cycle ($\Delta_{i,j} < \Omega$). Si les sous-périodes partagent un

diviseur commun, les threads peuvent être ordonnancés de façon qu'ils ne partagent jamais un cycle ($\Delta_{i,j} \geq \Omega$). Si une paire de thread doit occuper un même cycle, une section du cycle doit être réservée pour chaque thread. Ainsi, la notion d'une section est née, et chaque section sera associée à un nombre premier. Ensuite, dans chaque section, chaque thread peut avoir un décalage interne qui permettrait que plusieurs threads occupent la même section et le même cycle sans chevauchement.

En résumé, GCD+ :

1. associe chaque thread à une section selon la factorisation en nombres premiers de sa sous-période ;
2. décide quel cycle le thread va occuper ;
3. calcule le décalage interne nécessaire pour empêcher le chevauchement des threads de la même section qui occupent le même cycle ;
4. calcule la taille de chaque section en prenant en compte les décalages internes et le temps d'exécution de chaque thread ;
5. et, enfin, calcule le décalage final à partir des informations précédentes.

Pour que cette méthode puisse bien fonctionner, Ω doit être suffisamment large pour contenir toutes les sections. Même si la présence d'un PGCD global assez large puisse paraître une contrainte forte, cette contrainte est satisfaite dans une partie importante des cas d'usage réels, vu que leurs threads ont des périodes très larges par rapport à l'unité de temps du système, et que le choix des périodes est normalement fait par un humain, qui aura la tendance d'arrondir les chiffres. GDC+ exploite cette propriété pour calculer des décalages mieux adaptés à la situation, mais elle fournira aussi une proposition de décalages même si Ω n'est pas plus large que la somme de la taille des sections.

Pour tester la méthode, un générateur de jeux de threads aléatoire a été développé de façon à simuler des choix humains des valeurs des périodes. Basé sur la configuration des périodes des messages de télémétrie Paparazzi, l'algorithme extrait la probabilité qu'un tel facteur premier apparaîtra dans une période et génère des valeurs des périodes considérant ces probabilités. Une génération aléatoire des valeurs d'utilisation (C_i/T_i) est faite pour que les valeurs des temps d'exécution puissent être calculées, tout en gardant une utilisation totale fixée à une valeur fournie par l'utilisateur.

Les jeux de threads sont donnés à quatre algorithmes de génération de décalage : la méthode Goossens, la méthode pour les messages CAN, la méthode Paparazzi et, finalement, GCD+. Le temps de calcul est de l'ordre des millisecondes pour toutes les méthodes, donc leur efficacité a été choisie comme critère d'évaluation. Pour évaluer l'efficacité, le retard maximal auquel chaque thread subit (dans une simulation du système) sont comparés dans leur état brut ou normalisés soit par la période du thread, soit par le temps d'exécution maximale parmi les autres threads, soit par son propre temps d'exécution. L'ordonnançabilité est aussi évalué, en considérant que les threads ont des échéances implicites.

Les résultats montrent que GCD+ réduit sensiblement les retards subis par les threads, et permet une augmentation importante du respect aux échéances par rapport aux autres

méthodes, surtout quand le nombre de threads dans un système augmente. Ces résultats ont été confirmés par un cas d'étude réel sur la télémétrie Paparazzi, où GCD+ a pu empêcher la perte des messages due au débordement de la file d'attente.

GCD+ peut être utilisé pour les messages dans un lien de communication, mais aussi pour des threads dans un exécutif cyclique. Une possible extension de GCD+ pourrait considérer des précédences entre les threads, exprimée selon la technique SPC. En plus, GCD+ peut être adapté pour gérer plusieurs liens de communication à la fois, ce qui est le sujet du prochain chapitre.

Chapitre 4 - GCD# : Ordonnancement des Réseaux

Pour permettre l'ordonnancement statique des trames dans un réseau commuté, GCD+ a été étendu : l'algorithme GCD# calcule un ensemble de décalages initiaux avec le but de réduire la latence pour chaque flux de messages.

Un flux de messages est une structure analogue à un thread, où chaque message est analogue à un job, et chaque port de sortie d'un commutateur est analogue à un processeur monocœur. En modélisant les ports de sorties et les chemins parcourus par les flux de messages, nous pouvons construire un graphe qui représente le réseau. Le retard technologique entre la réception d'un message et sa retransmission doit être pris en compte.

La condition dans laquelle la latence d'un message est égale simplement au retard technologique est appelée Condition de Latence Minimale. La condition de latence minimale n'est pas atteinte dès qu'un message doit attendre, dans la file de sortie, pour la fin de l'envoi d'un autre message, ce qui augmente la latence. GCD# cherche des valeurs de décalages de façon à attendre la condition de latence minimale.

Pour ce faire, GCD# considère que chaque message, dans chaque port de sortie, subit un retard constant, appelé Enregistrement et Réémission (S&F). En plus, en considérant le chemin parcouru par chaque message, la logique de GCD+ peut être appliquée.

D'abord, GCD# calcule le PGCD global Ω. Ensuite, elle associe chaque flux de messages à une section selon les facteurs premiers de la sous-période de chaque flux. Puis, le cycle est choisi pour chaque flux, en considérant que seulement les flux qui se croisent peuvent interférer l'un avec l'autre directement. Le décalage interne est calculé en considérant le retard technologique différentiel que les trames peuvent subir selon leurs chemins. Après que tous les flux sont considérés, les tailles des sections sont calculées. Finalement, la valeur finale du décalage de chaque flux est calculée.

Par contre, pour chaque choix, une sous-heuristique doit être mise en place, vu que l'algorithme ne disposera pas de toutes les informations nécessaires pour déterminer le meilleur choix à prendre. Dans cette version de l'algorithme, des sous-heuristiques classiques ont été implémentées, ce qui laisse une marge d'amélioration à l'algorithme. En plus, les tailles des sections sont calculées de façon conservative, et une analyse plus précise pourrait réduire le

pessimisme.

Cependant, la méthode GCD# a été capable de proposer des décalages des flux de messages pour un cas d'étude avec 100 flux, 15 commutateurs et plus de 30 nœuds émetteurs des flux : le réseau Orion. GCD# a proposé les décalages dans l'ordre de millisecondes, et la somme de la taille des sections était inférieure à 20% de la taille du PGCD global, ce qui indique que, dans le modèle d'exécution proposé avec S&F constant, aucun message ne doit attendre l'envoie d'un autre message, et la condition de latence minimale est atteinte. Des outils de programmation par contraintes et d'optimisation ont été également utilisés pour le même cas d'étude, mais ses résultats n'ont pas été satisfaisants, vu un temps impraticable pour aboutir à un résultat.

GCD# s'est présentée comme une méthode applicable aux grandes échelles, contrairement aux méthodes qui explorent tout l'espace de solutions. La méthode peut toujours être améliorée par rapport à la façon qu'elle résout les sous-problèmes, mais elle se présente déjà comme une solution compétitive en comparaison avec les méthodes de l'état de l'art.

Chapitre 5 - Contexte en Ingénierie Dirigée par des Modèles

Pour aider à maîtriser la complexité des systèmes embarqués des drones, l'Ingénierie Dirigée par des Modèles (MDE) peut fournir un important appui. La MDE est une notion qui formalise l'abstraction des systèmes complexes, de façon à fournir un outillage puissant pour gérer leur complexité. Dans la MDE, tout modèle doit être conforme à un méta-modèle, qui doit lui-même être conforme à un méta-méta-modèle, et ainsi de suite, jusqu'à ce que l'on trouve un modèle qui est conforme à lui-même. Ces méta-modèles, quand ils utilisent des concepts spécifiques d'un domaine d'application, peuvent être vus comme des Langages de Modélisation Spécifiques au Domaine (DSML). Chaque DSML contient une perspective unique des objets modélisés.

Les instances des DSMLs peuvent subir des transformations : un modèle A qui implémente le DSML A peut être utilisé par un modèle de transformation pour générer le modèle B, conforme au DSML B, qui représente le même objet de A. Cette transformation peut être faite de façon automatique, ce qui représente un outil puissant de changement de perspectives à partir des modèles. Elles permettent par exemple de réaliser une analyse des propriétés d'un modèle pour que l'objet relatif au modèle soit validé selon une perspective de fonctionnement, représentée par le DSML utilisé.

Un exemple de DSML est AADL, un langage de représentation des architectures de hardware et software des systèmes embarqués. Ces concepts incluent celui d'un processeur, un bus, une mémoire, un procès, un thread, une donnée, des ports d'entrée et de sortie, etc. Une alternative à AADL est MARTE, une extension du langage de modélisation unifié UML, qui permet la description des systèmes embarqués.

De façon similaire à la MDE, des outils de conception des missions de drones existent pour

faciliter l'adaptation des robots (y inclus des UAVs) pour des différents contextes, comme FlyAQ, RoboChart, RobMoSys, ReApp, RobotML, BRIDE, SmartSoft, etc.

De plus, des protocoles de communication sont un important outil pour modulariser et diviser la complexité du système, comme ROS, ROS2 (qui utilise DDS), ABI et MAVLink. ROS et ROS2 utilisent le concept de Topic pour représenter un lien de communication, où un nœud (ROS) ou participant (ROS2) peut publier ou auquel il peut s'abonner. Cela permet une certaine indépendance des modules les un des autres, et il peuvent même se connecter et se déconnecter pendant l'exécution du système. ROS2, de son côté, utilise le protocole de communication DDS comme une sous-couche, ce qui lui confère aussi une décentralisation qui n'est pas présente dans ROS. ABI est le middleware ad-hoc de Paparazzi, développé pour un usage seulement interne à l'autopilote, mais qui permet aussi à ses modules d'être indépendants des autres modules. MAVLink, par contre, est un protocole de communication de télémétrie entre le drone et la station sol. Avec ce protocole, les commandes et les informations transmises par et pour le drone peuvent être standardisées, et les UASs qui l'utilisent sont plus personnalisables.

La méthode Arcadia a aussi inspiré les contributions de cette thèse. Basée sur des langages comme AADL et SysML, elle promeut la création de différents modèles qui représentent le même système, mais à partir des perspectives différentes : opérationnelle, fonctionnelle, logique et physique. Ces représentations sont liées et permettent une traçabilité des éléments entre les modèles.

Enfin, le projet RobMoSys a rassemblé un outillage qui permet le développement des systèmes robotiques à partir des modèles. Certains de ces outils permettent même la génération automatique de code à partir d'un modèle, ce qui peut être interprété comme une transformation de modèles.

Chapitre 6 - RoBMEX

Pour une première approche, nous utilisons l'autopilote comme une boite noire, un composant disponible en marché (COTS). Ce composant, déjà validé et disponible pour utilisation, fournit une interface qui sera utilisée pour sa personnalisation.

Deux utilisateurs sont prévus pour cette approche : l'utilisateur potentiel des drones (utilisateur final) qui n'est toujours pas familiarisé avec la programmation des systèmes embarqués, mais qui veut adapter le comportement d'un drone à son application, et le développeur des comportements complexes (expert), à l'aise avec la programmation robotique, mais qui n'est pas encore à l'aise avec la programmation des autopilotes des drones spécifiquement. Ces deux utilisateurs sont liés par le concept d'une mission, une suite d'actions exécutées par le drone. L'utilisateur final veut composer une mission à partir des blocs déjà prêts, et l'expert peut développer ces blocs individuellement.

Pour fournir à ces deux types d'utilisateurs un outillage qui facilite leur travail, nous

proposons la méthodologie RoBMEX, basée sur les technologies ROS et MAVLink. Le but de RoBMEX est de créer un modèle de comportement du drone à partir d'une bibliothèque d'éléments développés précédemment, de façon que ce modèle puisse générer automatiquement le code d'un nœud ROS qui communique avec un drone via MAVLink (en utilisant le nœud MAVROS comme un pont entre le système ROS et le drone), ou le code d'une application embarquée dans une carte accompagnatrice de la carte de l'autopilote.

Pour cela, RoBMEX est divisé en trois DSMLs : ROSProML (un langage de définition des processus qui utilisent les structures de données de ROS comme base), ROSMiLan (qui relie des instances de ROSProML dans une structure qui représente une mission), et ROSModL (qui modélise un système ROS).

ROSMiLan est responsable pour l'instanciation d'un arbre de comportement, où une séquence principale d'actions peut contenir des sous-séquences qui s'exécutent en parallèle. L'utilisateur final est l'utilisateur ciblé pour ce langage, et cet utilisateur peut personnaliser un comportement complexe d'un drone sans avoir les connaissances de programmation requises pour écrire le code du comportement.

L'élément fondamental de ces arbres de comportement est un objet instancié par ROSProML, qui sera créé par l'expert de l'algorithme modélisé. Même s'il est l'expert de l'algorithme, il peut ne pas connaître les interfaces spécifiques aux technologies associées à l'autopilote de drone en question, mais il peut toujours créer le modèle de cet algorithme dans ROSProML. Ce modèle servira comme un élément d'une librairie qui sera à disposition de l'utilisateur finale, à partir de laquelle il pourra construire son arbre de comportement (la mission du drone).

Enfin, ROSModL est utilisé comme une étape importante de la transformation de modèles responsable pour générer un système ROS à partir d'une instance de ROSMiLan. Cette transformation de modèles est basée sur Accéleo.

Pour démontrer la faisabilité de RoBMEX, un cas d'étude simple a été analysé d'un robot qui doit lire une distance à un obstacle et, à partir d'une comparaison de cette distance à une distance minimale, décider s'il doit avancer ou reculer. Ce cas a pu être modélisé grâce à l'implémentation de RoBMEX faite sur Ecore (pour Eclipse).

Dans un premier cas, l'instance de RoBMEX communiquera avec l'autopilote via messages MAVLink. Pour ce faire, un nœud MAVROS doit être présent dans le modèle du système ROS. Pour ce cas, un code compilable (sans erreurs de compilation) a été généré, même si le code qui détermine le comportement de chaque fonction n'ait pas été produit, vu que la génération de code n'est toujours pas complètement définie.

Dans un deuxième cas, la génération d'un code qui communique directement avec les pilotes des capteurs et actionneurs du drone a été explorée. Cette génération n'est pas dépendante de MAVROS, mais elle requiert que les instances de ROSProML puissent modéliser correctement une communication avec les pilotes. La génération de code a également abouti sans erreurs de compilation, mais la communication avec les pilotes n'a pas été modélisée.

La génération de code de RoBMEX est toujours un travail en développement, ainsi que son interface graphique pour l'instanciation des modèles, basée sur Sirius. Cependant, RoBMEX a le potentiel de réduire les efforts de personnalisation des drones sans changer le code embarqué dans l'autopilote. RoBMEX pourra être adapté pour se baser sur ROS2 au lieu de ROS, ce qui permettrait aussi une communication plus adaptée aux autopilotes qui s'intègrent plus facilement avec DDS, comme PX4. Il est également possible d'adapter RoBMEX pour intégrer ses modèles aux outils d'analyse temps-réel, et pour qu'il puisse aussi gérer les cas de flottes de drones.

Chapitre 7 - Architecture de Référence

Pour permettre une adaptation des autopilotes des drones et une standardisation qui faciliterait la réutilisation des modules du logiciel embarqué, nous proposons dans ce chapitre une architecture de référence pour les autopilotes des drones. Cette architecture peut être vue par trois perspectives différentes : fonctionnelle, logicielle et temps-réel. Avec l'adoption de cette architecture, les autopilotes pourront être considérés comme des composants modifiables (MOTS), personnalisables, mais de telle sorte que les modifications puissent être isolées et que ses impacts sur la sûreté soient minimisés. L'architecture proposée est basée sur les architectures déjà présentes dans le marché.

L'architecture fonctionnelle présente dans les trois autopilotes recherchés et le contrôle en cascade, où le drone est stabilisé par des boules de contrôle qui s'exécutent dans une pile. La stabilité directement contrôlée est celle de la vitesse angulaire, et une boucle de contrôle de position angulaire s'exécute sur la boucle de vitesse angulaire, en lui fournissant la valeur cible à stabiliser. De façon analogue, les boucles d'accélération linéaire, puis vitesse linéaire, et enfin position dans l'espace s'exécute sur la pile de contrôle.

Des modules fonctionnels peuvent être isolés pour représenter la navigation – l'interprétation des données captées par le drone pour estimer sa position linéaire et angulaire, ainsi que ses dérivées — et la perception – l'interprétation des données des capteurs pour recueillir des informations des alentours. Ces modules sont responsables pour envoyer l'état actuel du drone aux modules de contrôle en cascade pour qu'ils appliquent les corrections nécessaires.

De plus, un module de guidage analyse les informations des modules de perception et navigation pour gérer la trajectoire et la mission du drone. Ce module est responsable pour l'autonomie de l'UAV. D'autres fonctions peuvent être rajoutées autour de cette structure centrale, mais le noyau fonctionnel de l'autopilote garde cette structure.

Du point de vue logiciel, une stratégie souvent utilisée est l'abstraction de la gestion du hardware. La couche HAL d'abstraction des pilotes semble fondamentale pour que le code de l'autopilote soit réutilisable. Au-dessus de cette couche, le code de l'autopilote doit aussi être modulaire, de sorte que chaque fonction soit isolée dans son module logiciel. Un bus logiciel (middleware) peut contribuer fortement à l'isolation nécessaire.

D'un point de vue temps-réel, quelques implémentations des autopilotes utilisent des systèmes d'exploitation temps-réel (RTOS), avec un ordonnanceur et la possibilité de préemption. Par contre, d'autres implémentations n'utilisent pas un système d'exploitation (surtout les implémentations qui sont nées il y a plus longtemps). Celles-ci utilisent un exécutif cyclique classique pour implémenter les fonctions du noyau de l'autopilote. Par contre, même les implémentations avec RTOS utilisent un seul thread pour implémenter l'autopilote – ou, au pire, deux : un thread pour la boucle rapide (attitude) et l'autre pour la boucle lente (position). La sous-division du noyau de l'autopilote semble générer des problèmes de communication entre les threads qui demanderaient un effort plus important que les avantages qui pourraient provenir de ce type d'implémentation. Par conséquent, le noyau de l'autopilote doit se limiter à, au plus, deux threads. Des modules comme la gestion de la charge utile, la gestion de la santé du drone, la gestion du stockage des données et la communication peuvent être dissociés du noyau dans des threads moins prioritaires.

Pour implémenter l'architecture de référence, nous ne proposons pas une implémentation spécifique, mais un guide de comment étendre et adapter des architectures existantes avec les concepts fondamentaux à cette approche. La liste de concepts à intégrer dans les implémentations est la suivante :

1. Fonction
2. Entrées et sorties des fonctions
3. Communication intra-thread
4. Décomposition hiérarchique des fonctions
5. Logique dans une fonction
6. Thread
7. Ordonnancement
8. Communication inter-thread
9. Ordonnanceur statique au niveau des threads (par exemple, exécutif cyclique)
10. Procès (isolation mémoire)
11. Partitions
12. Plateformes (hardware)
13. Bus logiciel

Dans un cas pratique, l'architecture logicielle de Paparazzi a été analysée du point de vue de l'architecture de référence. Des chaines fonctionnelles ont été extraites et une relation entre la fréquence des capteurs et l'ordonnancement de l'autopilote a été mise en avance. Basée sur cette relation, l'implémentation d'une version de GCD+ qui considère les précédences pourrait améliorer l'exécution de l'autopilote en synchronisant les modules de façon à éviter une surcharge du cycle de l'exécutif cyclique.

Conclusion

Cette thèse, partie du projet COMP4DRONES, évalue le contexte du développement des drones autonomes en Europe. Les drones peuvent effectuer diverses tâches fastidieuses ou

dangereuses pour les humains, mais leur utilisation reste limitée à des domaines spécifiques. La législation actuelle ne prend pas en compte les aspects logiciels embarqués, rendant leur utilisation coûteuse et risquée. La thèse vise à combler les lacunes de connaissances pour créer des cadres technologiques qui favorisent le développement des drones en Europe.

La thèse analyse des pilotes automatiques de drone en tant que systèmes embarqués temps réel pour mettre en évidence les aspects de sûreté. Un nouvel algorithme, GCD+, améliore la planification de l'exécution cyclique, augmentant la sécurité du drone, et des adaptations comme GCD# sont présentées. L'adaptabilité de l'architecture du pilote automatique est également évaluée pour permettre une personnalisation efficace des drones.

Deux approches de personnalisation sont abordées : l'utilisation de pilotes automatiques prêts à l'emploi ou leur personnalisation. RoBMEX, un langage de modélisation basé sur ROS, est développé pour faciliter l'instanciation des missions de drones en utilisant la première approche. Une architecture de référence est proposée pour promouvoir l'interopérabilité entre modules de différents fournisseurs, visée à faciliter la deuxième approche.

En résumé, cette thèse vise à réunir la modularité et la sûreté dans le développement des drones autonomes pour créer des conceptions personnalisables et sûres, favorisant ainsi leur plein potentiel en Europe. Il s'agit d'un travail toujours ouvert, et d'autres projets peuvent suivre les développements présentés ici, comme pour lier les exigences de bas niveau (évoquées dans cette thèse) aux exigences de haut niveau.

Vers une Conception Modulaire et Sûre des Autopilotes des Drones Autonomes

Résumé — Dans cette thèse, nous cherchons à réduire l'écart entre la personnalisation des drones autonomes et leur sûreté de fonctionnement en analysant l'état de l'art actuel de la sûreté des systèmes embarqués temps réel. Les pilotes automatiques de drones sont étudiés dans leur cœur, ainsi leur dynamique d'exécution est mise en évidence et des améliorations de sûreté sont proposées. L'exécutif cyclique au cœur de la plupart des pilotes automatiques open-source est comparé à un système offset-free, et l'algorithme GCD+ est suggéré comme outil pour calculer les décalages pour de tels systèmes. Plus tard, une adaptation de cet algorithme, appelée GCD#, est suggérée pour calculer les décalages pour les flux de messages dans les réseaux commutés. Ensuite, une approche basée sur des modèles est proposée pour augmenter l'adaptabilité des pilotes automatiques de drones en tant que composants prêts à l'emploi : RoBMEX est présenté comme un cadre de modélisation pour adapter les drones à diverses missions, basé sur ROS et le protocole de communication MAVLink. Enfin, la modularité est évaluée en étudiant les architectures de pilote automatique actuellement utilisées en ce qui concerne leurs modules fonctionnels, et des lignes directrices sont établies sur la base d'une architecture de référence pour permettre la personnalisation des pilotes automatiques de drones sans perdre de vue leur sûreté de fonctionnement.

Mots clés : Drones. Fiabilité. Ingénierie dirigée par les modèles. ROS (système d'exploitation des ordinateurs). Systèmes embarqués (informatique). Temps réel (informatique). Autonome. Micro Air Vehicle Link (MAVLink). Unmanned Aerial Vehicle (UAV).

Towards a Safe and Modular Architecture for Autonomous Drone Autopilots

Abstract — In this thesis, we seek to reduce the gap between autonomous drone customization and safety by analysing the current State of the Art of real-time embedded system safety. Drone autopilots are studied at their core, so their execution dynamics are highlighted and safety improvements are proposed. The cyclic executive in the core of most open-source autopilots is compared to an offset-free system, and the GCD+ algorithm is suggested as a tool to calculate offsets for such systems. Later, an adaptation of that algorithm, called GCD#, is suggested to calculate offsets for message flows in switched networks. Then, a model-based approach is proposed to increase the adaptability of drone autopilots as Components Off-the-Shelf: RoBMEX is presented as a modelling framework to adapt drones to various missions, based on ROS and the MAVLink communication protocol. At last, modularity is assessed by studying currently used autopilot architectures with respect to their functional modules, and guidelines are made based on a reference architecture to allow for drone autopilots to be customized without loosing sight of safety aspects.

Keywords: Drone aircraft. Fiability. Embedded computer systems. Real-time data processing. Autonomous. Micro Air Vehicle Link (MAVLink). Model-driven engineering. Robot Operating System (ROS).