A generalized asynchronous method preserving the data quality of real-time embedded systems : Case of the PX4-RT autopilot

Evariste Ntaryamira

Thèse de doctorat de

SORBONNE UNIVERSITÉ

École doctorale Informatique, Télécommunications et Électronique (Paris)

Spécialité

Informatique

Présentée par

Evariste NTARYAMIRA

Pour obtenir le grade de

DOCTEUR DE SORBONNE UNIVERSITÉ

Sujet de la thèse

Une méthode asynchrone généralisée préservant la qualité des données des systèmes temps réel embarqués

Cas de l'autopilote PX4-RT

---

soutenue le: 07/05/2021

Membres du jury

Mme. Liliana Cucu-Grosjean	Directrice de thèse
M. Cristian Maxim	Co-encadrant
M. Frédéric Boniol	Rapporteur
M. Laurent George	Rapporteur
Mme. Alix Munier	Examinatrice
M. Julien Forget	Examinateur

Contents

Contents 2

I CONTEXT AND STATE OF THE ART 1

1 The real-time domain 3

1.1 Introduction . . . 3

1.2 Real-time systems . . . 5

1.2.1 Uniprocessor vs Multiprocessor platforms . . . 7

1.2.2 Scheduling algorithms classifications . . . 8

1.2.3 The tasks scheduling transition state diagram . . . 10

1.2.4 Real-time tasks precedence constraints . . . 11

1.3 Resource sharing and arbitration mechanisms . . . 12

1.4 Real-time data properties constraints . . . 16

1.4.1 Towards the functional chain formation . . . 17

1.4.2 Task-to-task data properties . . . 18

1.4.3 End-to-end data properties . . . 28

2 Unmanned aerial vehicles (UAVs) 33

2.1 Related work . . . 33

2.1.1 Communication and control system in UAVs . . . 35

2.1.2 Coupling effect in UAVs . . . 37

2.2 The PX4 autopilot . . . 39

2.2.1 The Flight Stack . . . 40

2.2.2 The middleware architecture . . . 42

2.3 The thesis context . . . 51
2.3.1 The PX4-RT system . . . 52
2.3.2 The thesis goals . . . 53
2.3.3 Identifying data constraints covered in this thesis . . . 53

II CONTRIBUTIONS AND RESULTS 58

3 Modeling and formalization 60
3.1 The tasks system model . . . 60
3.2 The communication model . . . 62
3.2.1 A basic organization of the buffer . . . 62
3.2.2 Message characterization . . . 65
3.2.3 Data sample characterization . . . 67
3.3 The communication graph . . . 69
3.3.1 Shared variables management . . . 71
3.3.2 Inter-task communication relations . . . 72

4 Local consistency constraints 74
4.1 Computing the buffers optimal size . . . 76
4.2 The sub-sampling rate mechanism . . . 80
4.2.1 Introduction . . . 81
4.2.2 Formalization and implementation . . . 82
4.3 On the verification of local consistency constraints . . . 88
4.3.1 Data freshness . . . 88
4.3.2 Data local coherence . . . 89
4.3.3 Data consistency . . . 89

5 Global consistency constraints 91
5.1 CPU and memory storage requirements . . . 93
5.2 Spindle modeling and formalization . . . 94
5.3 The solution overview . . . 99

5.4 Verifying the global consistency constraints 102
5.4.1 The last reader tags mechanism 102
5.4.2 Computing the spindle source buffer size 105
5.4.3 Spindle chains propagation delays 109
5.4.4 The scroll or overwrite mechanism 115

6 Conclusions and perspectives **122**

Bibliography **126**

List of Figures

1.1 The main temporal characteristics of a real-time task. 5
1.2 The tasks scheduling transition state diagram 10
1.3 Example of the communication order between tasks. 11
1.4 The state transition diagram in the presence of resource sharing constraints . . 13
1.5 Effects of the mutual exclusion constraint. 14
1.6 Extending the model on Figure 1.5 with an intermediate priority task 15
1.7 Data consistency motivating example 20
1.8 Violation of consistency with other variables example 20
1.9 Violation of consistency in value example 20
1.10 Maintaining data consistency using mutual exclusive resource 20
1.11 Example of an explicit access to the shared register. 26
1.12 Example of an implicit access to the shared register. 27
1.13 Example of k diverging functional chains. 29
1.14 Example of k converging functional chains. 29
1.15 Example of the `spindle propagation chains` 30
1.16 Data matching motivating example. 31

2.1 The PX4 High-Level Software Architecture 41
2.2 The PX4 flight stack diagram 42
2.3 The non-exhaustive list of the μORD topics in the local directory 44
2.4 The C/C++ header derived from the topic depicted in Figure 2.3 44
2.5 The `follow_target.msg` file content 45
2.6 Exploring the `follow_target.h` header content 45

2.7 The structure of the orb metadata . . . 46
2.8 The μORB data state transition diagram . . . 49
2.9 Data constraints and properties overview . . . 57

3.1 The task execution time is the sum `r+p+w` . . . 62
3.2 An example of a circular buffer shared between two tasks . . . 63
3.3 Tasks used to check the age of the data within the circular buffer . . . 64
3.4 Example displaying the age of the data within the circular buffer . . . 64
3.5 Example of input and output messages for the task `position_ctrl` . . . 66
3.6 An example of a multi-source message . . . 66
3.7 The `vehicle_attitude_setpointp` message . . . 67
3.8 Example of a data double stamping process . . . 68
3.9 An example of the communication graph . . . 70
3.10 Example of the message domains . . . 71
3.11 An example displaying communication variables shared between adjacent tasks 71
3.12 Example showing the memory space allocated to the data variable $\mathcal{D}$ (the first bit of the first byte). $\mathcal{D}$ occupies contiguous addresses . . . 71
3.13 A `linear relation` . . . 72
3.14 A `fork relation` . . . 72

4.1 Example of two buffers written by a same task but likely to have different sizes. 76
4.2 Example of a bi-dimensional multi-source buffer where each of the vectors may have different sizes. . . . 76
4.3 The sub-sampling rate mechanism expected results . . . 81
4.4 An example showing the sub-sampling mechanism result . . . 90

5.1 A balanced spindle example. . . . 95
5.2 An α-balanced spindle example. . . . 95
5.3 An unbalanced spindle example. . . . 96
5.4 A general configuration configuration of the spindle . . . 98
5.5 Major notations used in the spindle data propagation . . . 99
5.6 The global consistency constraints verification scheme . . . 102

5.7 Example showing the data tagging operation 104
5.8 Example displaying `SCI` value for a set of tasks 108
5.9 Example displaying the swt_c^{min} and swt_c^{max} values for each τ_c^{start} 112
5.10 A balanced spindle application example. 118
5.11 The global consistency verification process example 120

List of Algorithms

1 *The circular buffer data variable memory allocation* 80

2 *Implementation of the sub-sampling rate mechanism* 87

3 *Implementation of the last reader tags mechanism* 106

4 *Implementation of the scroll or overwrite mechanism* 118

5 *Data matching algorithm for the spindle sink task* 121

List of Symbols

Symbol	Description
$\mathcal{T}$	A real-time task system
τ_i	A specific task in $\mathcal{T}$ such that $1 \leq i \leq n$ with n the numbers of tasks
c_i	Best-case execution time of a task τ_i
C_i	Worst-case execution time (WCET) of a task τ_i
T_i	Period of a task τ_i
D_i	Deadline of a task τ_i
O_i	Offset of a task τ_i
R_i	Worst-case response time τ_i
θ_i	Scheduling state of τ_i
ϖ_i	Communication state of τ_i
$\mathcal{H}$	Hyper-period of the tasks of $\mathcal{T}$
$\mathcal{M}$	The set of messages enhancing the communication between the tasks in $\mathcal{T}$
m	A given message in $\mathcal{M}$
$\mathcal{M}_i^{In}$	The set of τ_i input messages
$\mathcal{M}_i^{Out}$	The set of τ_i output messages
∇	A communication resource. In this thesis a communication resource is a circular buffer, which is a communication resource characterized by the tuple $\nabla(\mathcal{D}, size, head, tail)$. There is a communication resource for each message m and for the message m the corresponding communication is written as ∇^m.
$\mathcal{D}$	An array containing the communication data (input data for consumer tasks or output data for the producer task). An array containing the data

	carried by the message m is denoted by $\mathcal{D}^m$.
$size$	For a message m, $size$ is the maximum number of different data copies that can be hold into $\mathcal{D}^m$.
∂^m	Each copy of the data, referred to as `data sample`, is denoted by ∂^m. Each data sample is constituted by the values assigned to each of the variables forming the structure of the corresponding message. Each data sample occupies a single slot (cell) in $\mathcal{D}$.The slots are numbered from $0 \rightarrow size-1$.
$head(\nabla^m)$	An integer indicating, to the running job of a given producer task of a message m, into which slot (in $\mathcal{D}^m$) to write the output result at the end of the execution.
$tail(\tau_i, \nabla^m)$	As for the tasks reading from $\mathcal{D}^m$, each of them owns an $tail$ value indicating from which slot of $\mathcal{D}^m$ to read. So, $tail(\tau_i, \nabla^m)$ is the reading position of the task τ_i from the communication resource ∇^m.
$\mathcal{G} = (\mathcal{T}, \mathcal{M}, \mathcal{E})$	The communication graph for the tasks in $\mathcal{T}$ through the messages in $\mathcal{M}$ where $\mathcal{E}$ is the edges linking the tasks to messages
$\mathcal{E}^m$	The domain of the message m which is made of the producer and consumer tasks with respect to the message m.
$\mathcal{E}^*$	A general case of a message domain where * can be any message.
$Release(p, i)$	The release time of the $(p)^{th}$ job of a task τ_i
$Start(p, i)$	The activation time of the $(p)^{th}$ job of a task τ_i
$End(p, i)$	The execution completion time of the $(p)^{th}$ job of a task τ_i
$hp(i)$	The set of higher priority tasks of task τ_i
$\mathcal{I}^p_{i,h}$	The interference induced by all the $hp(i)$ tasks over τ_i from the release until the activation time of the $(p)^{th}$ job of a task τ_i
$delay^{Release(q,i)}_{End(p,j)}$	The amount of time units elapsed between the execution completion of the $(p)^{th}$ job of τ_j and the release of the $(q)^{th}$ job of τ_i
$delay^{Start(p,j)}_{Release(q,i)}$	The amount of time units elapsed between the release of the $(q)^{th}$ job of τ_i and the activation of the $(p)^{th}$ job of τ_j
$\eta^p_{i_j,p-1}$	The number of data samples that can be written by τ_i within $delay^{Start(p,j)}_{Release(q,i)}$.
$\mathcal{S}^m_q(\tau_{src}, \tau_{sink})$	A spindle comprised of a number q of functional chain propagating the data

	samples related to the message m from the τ_{src} until τ_{sink}
τ_{src}	The spindle source task
τ_{sink}	The spindle sink task
$\mathcal{C}_{c:q}^{m}$	The c^{th} out of q spindle chains comprising $\mathcal{S}_{q}^{m}(\tau_{src}, \tau_{sink})$
τ_{c}^{start}	The second task belonging to the c^{th} spindle chain
τ_{c}^{last}	The second-to-last task belonging to the c^{th} spindle chain
$\Omega_{c:q}^{min}$	The smallest amount of time it can take for a data sample read from the spindle source buffer to propagate until the spindle sink task with respect to the c^{th} spindle chain.
$\Omega_{c:q}^{max}$	The largest amount of time it can take for a data sample read from the spindle source buffer to propagate until the spindle sink task with respect to the c^{th} spindle chain.

Part I

CONTEXT AND STATE OF THE ART

Chapter

1 The real-time domain

Chapter content

1.1 Introduction 3

1.2 Real-time systems 5

1.2.1 Uniprocessor vs Multiprocessor platforms 7

1.2.2 Scheduling algorithms classifications 8

1.2.3 The tasks scheduling transition state diagram 10

1.2.4 Real-time tasks precedence constraints 11

1.3 Resource sharing and arbitration mechanisms 12

1.4 Real-time data properties constraints 16

1.4.1 Towards the functional chain formation 17

1.4.2 Task-to-task data properties 18

1.4.3 End-to-end data properties 28

1.1 Introduction

The human life in the twenty-first century is surrounded by technology. From household to transportation, education to hobbies, and safety to sports, information technology plays a major role in daily activities. Social interaction, education and health are few examples of areas, where the rapid evolution of technology has had a major positive impact on the quality of life. Companies are increasingly relying on embedded systems to increase productivity,

efficiency and business value. In factories, the precision of robots tends to replace human versatility. Connected devices such as drones, autonomous vehicles, smart watches or smart houses have become increasingly popular in recent years, offering a variety of facilities with high productivity. However, their integration requires to guarantee that there will not be any catastrophic repercussions on the human life. To ensure these guarantees, the functioning of such systems must be predictable over time through the application of temporal constraints which must be verified to that end and we name here, the real-time systems. From this perspective, one of the properties expected from a real-time system is to respond timely to events from their environment.

Basically, these systems are composed of a large number of applications (tasks, programs) that are continuously communicating by propagating input data from one to the other. The communicating applications are organized into producers and consumers. Usually, communication between producers and consumers is done through communication registers or buffers. Producer applications use these communication channels to write output data while consumers use them to retrieve the required input data. A sequence of applications involved in the definition of a specific function is called a *functional chain*. The data is propagated through a *functional chain* and the delay between the time instant when the data is generated (at the beginning of the functional chain) and the time instant when it is consumed (at the end of the functional chain) is expected to be bounded. Associating a time delay to data transform them in real-time data.

A system of applications can be composed of several functional chains where some of the chains can propagate real-time data coming from different sub-systems or triggered by different clocks. Additionally, a functional chain may be composed of applications executing at different rates, resulting in under- or over-sampling of the data.

Evolving technologies within real-time embedded systems makes these systems intelligent in the sense that, at a given time, they are mandated to perform targeted functions autonomously. For example, autonomous vehicles have the ability to sense the surrounding environment and navigate by themselves while making driving decisions. This autonomy in decision making raises a particular interest on the data sharing management system between applications since the correctness of the decisions highly depends on the quality of input data.

Therefore, for these intelligent systems, it is not sufficient that all tasks are scheduled on time to be sure that the overall functioning of the system is correct. It should be noted that task scheduling consists in assigning to each application CPU time units necessary to complete its execution, an execution order when executed in presence of other tasks in accordance with a scheduling policy that are, often, priority-based. The schedulability of the task system is confirmed if all tasks complete their executions before their deadlines.

Finally, the overall functioning correctness depends both on the system schedulability and the quality of the input data, which is translated into a set of properties associated to these data.

1.2 Real-time systems

Real-time systems are computing systems that must react timely to events in the environment. Consequently, the correct behavior of these systems depends not only on the value of the computation but also on the time at which the results are produced. A reaction occurring too late could be useless or dangerous [1]. Real-time systems are composed by large number of software applications each of them playing its own role or function. In the real-time literature, the authors are also using terms like *thread*, *task*, *program* or simply *application*. Within this thesis we use the term *task* to name them. The basic model of the real-time system is introduced by Liu and Layland [2] and presented in Figure 1.1.

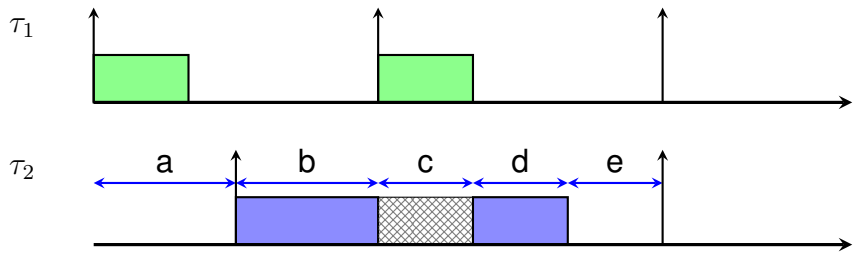

Figure 1.1: *The main temporal characteristics of a real-time task.*

The system of tasks denoted by $\mathcal{T}$ is composed by n tasks where each task $\tau_i \in \mathcal{T}$ is characterized by the tuples (O_i, C_i, T_i, D_i), with $\forall 1 \leq i \leq n$. O_i is the task **offset**. The task offset, also called the phase, is the time instant at which the first instance of the task is released. A task may execute without any offset and in this case we have $O_i = 0$. For

instance, in Figure 1.1, the offset of the task τ_1 is equal to 0, while the offset of the task τ_2 is equal to a. The time instant $r_2 = a$ is also known as the release time of the first instance of task τ_2. C_i is the task **worst-case execution time** (WCET). C_i represents the maximum time needed to execute the task non-preemptively. $C_2 = b + d$. The time instant at which a task finishes its execution is called the *completion time*, represented here by the time instant $t = a + b + c + d$. The amount of time between the task's release time and the task completion time is called the response time and denoted by R_i. Considering the same model, the response time of τ_2 is given by $R_2 = b + c + d$. The WCET is obtained after analysis, e.g. static or measurement-based, and most of the time the WCET is difficult to be accurately determined. At the opposite side, we encounter the notion of *best-case execution time (BCET)* which represents the minimum time length required for a task to execute. A detailed survey regarding the WCET static is provided by Wilhelm et al in [3] and a corresponding one on measurement-based analyses by Davis and Cucu-Grosjean [4]. T_i is the task's **period**. The period is the time between the releases of two consecutive instances of a periodic task. For sporadic task systems, the value of T_i is the minimum inter-arrival time between the releases of two consecutive instances of tasks. D_i is the task's **deadline**. The deadline indicates the time at which the task must have finished its execution for the correct functioning of the system. $\forall \tau_i \in \mathcal{T}$, if $D_i < T_i$ then τ_i is a *constrained deadline* task, $D_i = T_i$ τ_i is an *implicit deadline* task and $D_i > T_i$ τ_i is an *arbitrary deadline* task. The system can be extended with an other parameter called *jitter*. It is the deviation from the task's periodicity. For a task τ_i, the processor utilization, denoted by u_i is given by $\frac{C_i}{T_i}$ whereas the total utilization of all the set of task $\mathcal{T}$ is given by $\sum_{1 \leq i \leq n} \frac{C_i}{T_i}$, where n is the number of task comprising $\mathcal{T}$. The execution of the task system is said *synchronous* if all the tasks are released simultaneously. For constrained and implicit deadline tasks, the response time of the instance released synchronously with all other task instances is the largest. The impact of the execution of higher priority tasks over lower priority task τ_i is called the *interference* and denoted by I_i. The largest response time of τ_i is called the worst-case response time and denoted by R_i [5]. For synchronous constrained and implicit deadline tasks, the worst-case

response time is computed recursively as follows:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j \tag{1.1}$$

where $hp(i)$ is the set of higher priority tasks than τ_i and j one of them. The interference I_i from the j higher priority tasks is expressed within the part $\sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j$ of Equation 1.1.

Notice 1 *In this thesis we consider implicit deadline periodic tasks.*

1.2.1 Uniprocessor vs Multiprocessor platforms

Depending on the execution platform, the real-time system is called *uni-processor* if the tasks run on a single-processor computer and *multiprocessor* if the tasks run on a computer equipped with more than one processor. Multiprocessor systems are in their turn classified into *heterogeneous*, *homogeneous* and *uniform*. For *heterogeneous*, the processors are different and the execution rate of a task depends on both the processor and the task, *homogeneous* are made of identical processors which results in a same execution rate for all the tasks on all the processors. Finally, for *uniform* multiprocessor, the rate of execution of a task depends only on the speed of the processor. In this thesis, we consider *homogeneous multiprocessor* systems comprised of m processors.

1.2.1.1 Multiprocessor scheduling techniques classification

In Davis and al. [6] the authors classify real-time scheduling techniques in multiprocessor architectures according three categories: `partitioned`, `global` and `hybrid`. Under `partitioned scheduling`, tasks are assigned to processors, and the tasks within each processor are scheduled by a local scheduler. Regarding the `global scheduling`, all tasks are scheduled by a single scheduler, where a single queue is used to allocate tasks to processors. Tasks are dynamically allocated with inter-processor migration allowed. `Hybrid` scheduling combines the strengths of both partitioned and global scheduling. It can be further categorized into `semi-partitioned scheduling` and `clustering`. In semi-partitioned scheduling, most tasks are allocated to specific processors to reduce the number of migrations, while

other tasks are allowed to migrate to balance processors utilization. Clustering methods group a smaller number of faster processors into a cluster, and each cluster is scheduled with a different global scheduler. Among these scheduling techniques, partitioned scheduling is adopted and supported by domain-specific standards such as AUTOSAR [7] and commercial real-time operating systems (e.g. VxWorks, LynxOS and ThreadX).

Notice 2 *In this thesis we consider partitioned homogeneous multiprocessor platform. Shared resources are organized such that only tasks mapped on a same processor can communicate.*

1.2.1.2 The tasks models classification

Most of the research carried out on multi-processor systems focus on two main task models: *periodic models* and *sporadic models*. In both cases, each task τ_i generates an infinite number of successive jobs and, for simplicity, in the reminder of this thesis, we do not focus on specific jobs. Therefore, τ_i has the meaning of any of its jobs. We define the hyper-period as the least common multiple of the periods of all tasks and we denote it by $\mathcal{H} = lcm\left\{T_i\right\} \Big| i = 1, \cdots, n$. When the tasks are released simultaneously and have implicit deadlines, the interval $(0, H)$ is a feasibility interval for the task system $\mathcal{T}$ [8]. By feasibility interval we understand the smallest time interval such that if all deadlines are met within this interval, then all deadlines are met for the entire system. In the periodic task model, the jobs of a task arrive strictly periodically, spaced by a fixed time interval (period). In the sporadic task model, each job of a task can arrive at any time once a minimum interval time has elapsed since the arrival of the previous job of the same task. Task scheduling on a multiprocessor system involves solving the *task allocation* and *priority assigning* problems. The former consists in allocating a given task to a given task while the latter defines when and in what order each job should execute with respect to the jobs of other tasks.

1.2.2 Scheduling algorithms classifications

In Carpenter and al. [9] the authors classify the multiprocessor scheduling algorithms into *migration-based* and *priority-based*. Regarding the migration-based algorithms, there may be `no migration`; each task is allocated to a processor and no migration is permitted, *task-level*

migration; the jobs of a task may execute on different processors where each job can only execute on a single processor, and *job-level migration* when a single job can migrate to and execute on different processors with no parallel execution of a job permitted. As for the priority-based algorithms, we may list:

- *Fixed task priority algorithm*: It assigns the same priority to all the jobs of each task. To this category belong the rate monotonic algorithm (RMA) [2] and the Deadline Monotonic (DMA) [10] Algorithms. The RMA assigns priorities to tasks according to their request rates. The higher priority is assigned to the task with higher request rate; that is, with shorter period. For the DMA, the higher priority is assigned to the task whose value given $D_i - C_i$ is shorter.
- *Fixed job priority algorithm*: The jobs of a task may have different priorities, but each job has a fixed priority. The Earliest Deadline First (EDF) algorithm belongs to this category.
- *Dynamic priority algorithm*: A single job may have different priorities at different times. An example of such algorithm is the least laxity first (LLF).

A scheduling policy allowing jobs of higher priority tasks to suspend the execution of jobs of lower priority tasks in progress is called `preemptive priority-driven` or simply `preemptive` scheduling policy. The action of suspending the execution of the lower priority tasks by higher priority tasks is referred to as *preemption*. When a job continues its execution after being preempted by the execution of jobs of high priority tasks, it is said that this job *resumes* its execution. On the other hand, a scheduling policy which does not allow the preemption of the jobs of lower priority tasks by the jobs of higher priority tasks is accordingly referred to as `non-preemptive` scheduling policy. Preemptions by higher priority jobs are covered by the priority-driven preemptive scheduler [2] while satisfying timing constraints and precedence constraints if any.

Notice 3 *In this thesis we consider a preemptive fixed task priority scheduling policy.*

For its practical relevance the majority of automotive controls are designed with static priority-based job scheduling. The operating system standards OSEK [11] and AUTOSAR [7].

1.2.3 The tasks scheduling transition state diagram

The real-time system tasks are managed by a dedicated operating system called a real-time operating system, shortly *RTOS*. The tasks are given the right to execute based on the predefined criterion called scheduling policy. The set of rules that, at any time, determines the order in which tasks are executed is called a scheduling algorithm. Within real-time literature, tasks are, often, given execution priorities according to a given scheduling algorithm. The specific operation of allocating the CPU to a task selected by the scheduling algorithm is referred as dispatching. Depending on the task's execution state, a task can be in any of these states:

Sleeping: The task has completed the execution of its latest released job and it is waiting for a new release, thus, for starting a new execution, or, it has never been executed.

Ready: The task can potentially execute if no task of higher priority executes. Ready tasks are put into the waiting queue and re-dispatched once the CPU is available.

Running: The task is currently executing. The scheduling algorithm may allow or not the tasks of higher priority to suspend the running lower priority. Accordingly, the scheduling algorithm is said *preemptive* and *non-preemptive*, respectively. The preempted task is inserted into the *ready queue* and the next time that the preempted task returns into the running state it *resumes* from where it was at the preemption time instant. The time instant when the task gets ready for execution independently on the CPU availability is called the *release time*, the time when a task enters the running state is called the *activation time* while the time when all task's instructions are executed corresponds to the *completion time*.

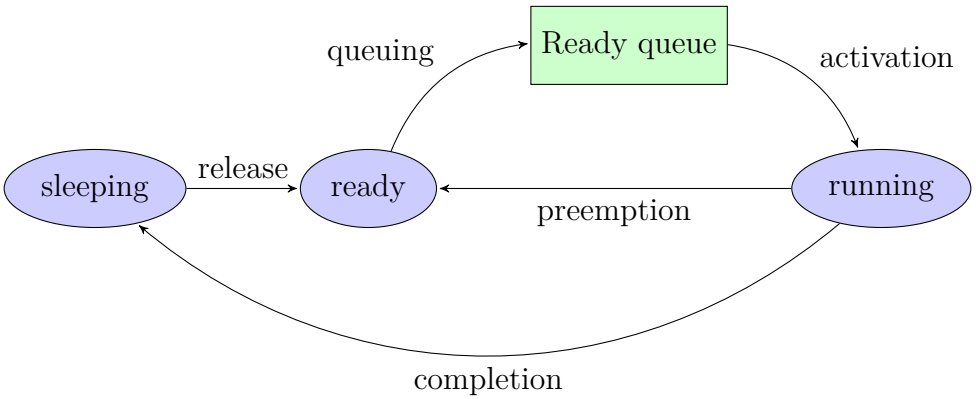

Figure 1.2: *The tasks scheduling transition state diagram*

1.2.4 Real-time tasks precedence constraints

Basically, real-time systems are composed of large number of tasks being in a permanent communication by sharing data in a way that the output of one task is used as input for another task and so on and so forth. The inter-tasks communications must fellow a strict order imposed by the designer. In other words, they must satisfy some communication ordering constraints. In the real-time literature, such constraints are widely referred to as precedence constraints and represented in the form of the directed acyclic graph (DAG). The directed acyclic graph is the graph $G = (V, E)$ where V is the set of tasks $\{\tau_1, \cdots, \tau_n\} \in \mathcal{T}$ and E is the set of edges. The precedence constraints are established between a pair of tasks both connected by the same edge. We use the right arrow $\rightarrow$ to represent the precedence relation between two tasks. For instance, the notation $(\tau_i, \tau_j) \in E$ indicates that tasks τ_i and τ_j are in a **precedence relation**. The task being at the left of the arrow is called a **predecessor** while the one at the right of the arrow is a **successor**. In the literature, a task that has no predecessor is often called a **sensor task** task, while a task that has no successor is called an **actuator task**.

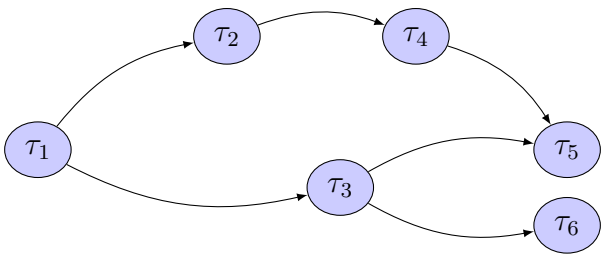

Figure 1.3: *Example of the communication order between tasks.*

In order to illustrate the precedence constraints between tasks, let us consider the model presented in Figure 1.3. Herein, we have τ_1 which is a predecessor to τ_2 and τ_3 ,τ_2 which is a predecessor to τ_4 and so on and so forth. The precedence constraints can be of two types: **scheduling order** and **communication order**.

An execution order constraint must be satisfied between two tasks if the execution of one task depends on the execution state of another. For instance, considering two tasks τ_i and τ_j such that $\tau_i \rightarrow \tau_j$, scheduling order constraints are required to be satisfied if the execution of τ_j can start only when τ_i has completed. In such situation τ_i and τ_j are said **dependent**

tasks. Reversely, if each task executes at its own pace based on its priority, such tasks are referred to as **independent tasks**.

On the other hand, the communication order constraint between two tasks must be satisfied if the precedence constraints between these tasks have the sense of the **producer-consumer relation**. Clearly, such constraints have the meaning of showing, for each tasks, the task producing the required input data. Accordingly, the task being at the left of the arrow is called a **producer task** while the one at the right side of the arrow is called a **consumer task**. The output data of the producer task is the input data for the consumer task, which is the producer for another consumer and so on and so forth.

1.3 Resource sharing and arbitration mechanisms

The tasks composing the real-time system are permanently communicating where some tasks produce the output data required by others tasks for their computations. The element through which the inter-task communications are performed is called *resource*. A resource can be a data structure, a set of variables, a main memory area, a file, a piece of program, or a set of registers of a peripheral device.

Notice 4 *In this thesis, by shared resources, we understand the data structures or other variables used to store the data into the memory; a shared resource has the meaning of a communication variable.*

Resources can be private or shared. A resource that can only be accessed by a single task is said **private resource**. Comparatively, if a given resource can be accessed by more than one task then this resource is called a **shared resource**. Depending on how shared resources are accessed by the competing tasks, shared resources are divided into **non-blocking** and **exclusive** resources. Speaking of the **non-blocking resources**, the competing tasks access the resource based on their scheduling priority no matter if the source is being used by other tasks. That to say that the resource can be accessed simultaneously by more than one task at a given time instant. For the **exclusive resource**, the situation is totally different; the competing tasks can't access the shared resource simultaneously. Each time a resource is

being used by one of the competing tasks this resource is locked for the rest competing tasks until this resource is released. Accordingly, the competing tasks are said to be in a mutual exclusion on a given resource. A set of instructions to be executed during the mutually exclusive time interval is called **critical section**. In Figure 1.4 we depict the state transition diagram for the set of tasks communication in the presence of the mutual exclusion on the shared resources.

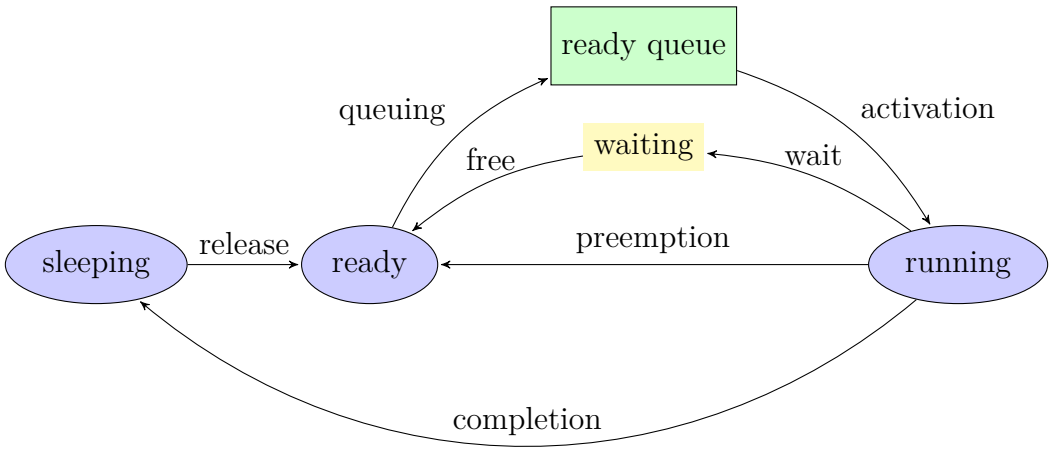

Figure 1.4: *The state transition diagram in the presence of resource sharing constraints*

Herein, supplementary to the scheduling state transition diagram presented in Figure 1.2, when the tasks communicate in the presence of mutual exclusion constraints on shared resources, another state referred to as `waiting` is introduced. Contrarily to the preemptions which are taken care by the scheduler, the mutual exclusion constraints are taken care by the operating system which arbitrate the access to such resources by means of semaphore per resource [12, 13]. Each exclusive resource is protected by two primitives: ***wait*** is placed at the beginning of the critical section to lock it when there is a task having entered this section. The primitive ***free*** is used to unlock the resource and is placed at the end of the resource critical section.

Any task attempting to access the shared exclusive resource is blocked until the completion of the concurrent task entered the critical section related to that resource. All tasks blocked on a given resource are inserted into the queue associated with this resource. In contrast to preemption, a higher priority task can be blocked by a lower priority task that has entered into the critical section regarding that exclusive shared resource. When the resource is available,

waiting tasks are not immediately granted an access to the resource. They are first inserted in the ready state and the scheduler decides to activate them according to their priorities.

For instance, let us consider two tasks τ_1 and τ_2 released simultaneously while competing on the mutual exclusion resource r such that τ_1 has a higher priority than τ_2. This situation is illustrated in Figure 1.5. Each white box with a number in it indicates the time period during which the task whose number is written in it was executing the critical section.

Herein, $\tau_{1,1}$ executes until its completion due to its high priority. The interval b corresponds to the time $\tau_{1,1}$ uses to execute the critical section instructions. The number in the white box corresponds to the index of the task holding the resource r at that instant.

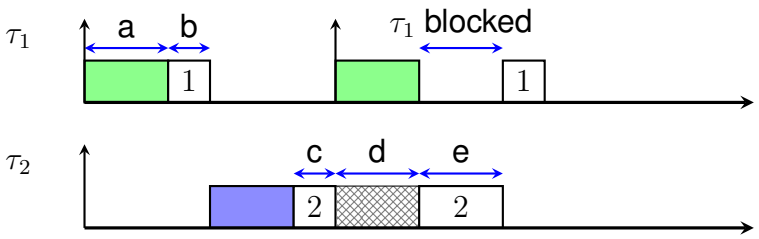

Figure 1.5: *Effects of the mutual exclusion constraint.*

The first job of τ_2 denoted $\tau_{2,1}$ (released after $\tau_{1,1}$ frees r) enters the critical section at the beginning of the interval c. Some time later $\tau_{1,2}$ preempts $\tau_{2,2}$ which still holds r. $\tau_{1,2}$ executes until it requests to access to r which is still blocked by $\tau_{2,1}$. Unfortunately, $\tau_{1,2}$ has to wait until r gets free. In this situation, $\tau_{2,1}$ resumes while $\tau_{1,2}$ is blocked. The blocking time is denoted by e. In the priority-driven real-time systems this situation leads to the so called ***priority inversion*** when the execution of a higher priority is blocked by the one of lower priority. The priority inversion phenomenon can lead to unpredictable system behavior or just force some jobs to miss their deadlines. To illustrate it, let us assume that τ_1 and τ_2 are released simultaneously. Normally, the worst-case response time corresponds to the response of the first job of each task [5].

$$\begin{cases} R_{1,1} = a + b \\ R_{1,2} = a + e + b \\ R_{1,1} < R_{1,2} \end{cases} \tag{1.2}$$

From the model presented in Figure 1.5, the blocking time is equal to e time units. Let us add to the system (Figure 1.6) a new task τ_m such that $priority(\tau_1) > priority(\tau_m) > priority(\tau_2)$, $O_m = 9$ and τ_m doesn't compete on r. According to its offset, we observe that it is released while $\tau_{2,1}$ is still inside the critical section related to r. Obviously, $\tau_{m,1}$ preempts $\tau_{2,1}$ and executes for the f time units. At its completion $\tau_{2,1}$ resumes and executes until its completion and only at that time $\tau_{1,2}$ resumes and completes.

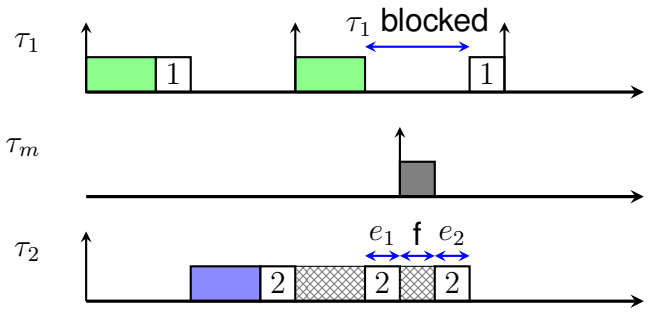

Figure 1.6: *Extending the model on Figure 1.5 with an intermediate priority task .*

Thus, a blocked job due to the mutual exclusion resource is blocked not only by the blocking job but also by all the jobs that can preempt the blocking job. For instance, considering the model in Figure 1.6, the blocking time of $\tau_{1,2}$ became $e + f$ where $e = e_1 + e_2$. Basically, real-time systems are composed of several tasks. The situation may end with unbounded response times if there are many mutual exclusion resources with many preemptions during the job blocking time on the resource. For instance, if the blocking job on the resource r_1 is preempted by a job competing on the resource r_2 which is also preempted while still holding this resource, and so on and so forth. This corresponds to the so called ***chained blocking***. Bounding the functional chained blocking time is a challenging task in complex systems. Hence, using arbitration mechanisms such as semaphores to protect the mutual exclusion resources may have negative impact on the system schedulability with prospect of leading to an unpredictable system behavior as they may provoke not only the priority inversion problem but also other possible deadlock situations [14–18].

1.4 Real-time data properties constraints

With the evolution of technologies, real-time embedded systems are getting more and more intelligent in the sense that, at some point, they acquire the capability to achieve targeted functions autonomously. For instance, the autonomous vehicles or drones are extended with capability of sensing the surrounding environment and navigating on their own by making driving decisions. The correctness of such decisions depends not only on the system schedulability but also on the quality of the data being used. The system schedulability is verified in order to guarantee that none of the jobs will miss its deadline whereas the quality of the data is expressed in terms of a variety of data properties that must be preserved throughout the inter-tasks communications for the correct functioning of the system. Speaking of the inter-tasks communications, in the literature a wide range of models, paradigms and policies are proposed.

Considering that predictability is a mandatory property for real-time systems, these communication models and policies can also be evaluated from this perspective. To that end, numerous constraints must be verified to guarantee the communication predictability as well as the correct functioning of the real-time embedded system. These constraints regard both the schedulability of the system and the quality of the data being used throughout the communications between the tasks. Several techniques are proposed in the real-time literature to analyze the system schedulability such as the task response time analysis technique [5], the processor utilization analysis technique [2] and the processor demand analysis technique [19]. These techniques efficiently handle the timing and precedence constraints while abstracting from the resource constraints even if the latter are on the basis of the scheduling problems such as priority inversion problem, the chained blocking or deadlocks formation [14–18]. Additionally to the scheduling problems, the lack of informations regarding the communication strategy plays an important role when it comes to maintain the properties of the data, especially the data consistency property. The constraints applied to this property require the protection of the data being used from any alteration/modification. To ensure this protection, one may choose to use mutual exclusive shared resources (i.e register of size one) which eventually leads to the memory space economy while still exposing the system schedu-

lability at risk (high consumption of the CPU resource). Conversely, it may be preferred to use buffers of larger sizes with an asynchronous non-blocking access while guaranteeing that none of the data will ever be corrupted. This strategy is safer in the sense that it does not induce unforeseen cpu time units compared to the utilization of mutual exclusive resource. Subsequently, the trade-off must be found between the `cpu overhead` or `memory overhead`. Last but not least, abstracting from the resource constraints verification may lead to a high variability of the tasks timing characteristics (when estimating/measuring the the tasks worst-case execution times) which ends up with significantly larger worst-case execution time. Therefore, the probability of appearance of a worst-case execution time is extremely low [20]. Approaches taking into account this probability have been the topic of last years research either by measurement-based reasoning for instance in [21] or static reasoning in [22, 23].

1.4.1 Towards the functional chain formation

The inter-tasks communications are carried out via shared variables such that the output data of one task is used as the input data for another task and so on and so forth until the expected action (function) is triggered. A sequence of the tasks, involved in the definition of a given function, forms the so-called `functional chain` [24–28] or the `cause-effect chain` [29–33]. In the reminder of this thesis we use `"functional chain"` to refer to such sequence of tasks that we denote by $\mathcal{C}$. Mainly, we model $\mathcal{C}$ as

$$\mathcal{C} = \tau_i \xrightarrow{r_i} \tau_{i+1} \xrightarrow{r_{i+1}} \cdots \xrightarrow{r_{n-2}} \tau_{n-1} \xrightarrow{r_{n-1}} \tau_n$$

where n is the number of tasks composing $\mathcal{C}$ and τ_i is any of the n tasks provided $1 \leq i \leq n$. r_i is the resource shared between the tasks τ_i and τ_{i+1} such that τ_i writes into r_i and τ_{i+1} reads from r_i. Throughout the functional chain the data written into r_1 is propagated by $n-1$ tasks before reaching the task τ_n being at the end of the functional chain and the propagation delay must be bounded [24, 31]. Hence, the constraints applied to these data must be satisfied not only by considering a given relation $\tau_i \xrightarrow{r_i} \tau_{i+1}$ where the tasks τ_i and τ_{i+1} compete on the resource r_i but also they must be applied to the entire functional chain

with the purpose to verify the end-to-end data temporal properties. Subsequently, different data properties must be ensured at different levels of the functional chain. For instance, it may be required that the content of the resource r_i should not be overwritten/modified by a job of τ_i if there is a job of τ_{i+1} still performing computations using the data read from r_i (`data consistency maintenance`) while, on the other hand, it can be required that the propagation delay of a data read from r_i until the last task of the functional chain must be within a bounded time interval (`end-to-end propagation delay`).

On this basis, we examine the data properties into two groups, `task-to-task` and `end-to-end` data properties. The first group concerns the data properties to be verified considering the pairs of adjacent tasks within the functional chain, while the second group concerns the data properties to be verified at the level of the functional chain. There may be a wide list of data properties falling into each of the categories. In the following section, we mention only those properties that are, directly or indirectly, relevant while expressing the contributions of this thesis.

1.4.2 Task-to-task data properties

For task-to-task data properties, we focus on the freshness and consistency of the data. Generally speaking, a data sample is said to be "fresh" if, at the time it is read by a job of the consumer task, it is not yet obsolete [34]; that is, it is still temporally valid. In Ricardo et al. [35] the authors define the worst freshness of a data sample as the maximum age that a data sample can attain. It gets older between the moment when the task producing it is activated and during the time interval when it can be used by the task having consumed it, i.e., until the moment when the next sample of the data is consumed by this task. In Forget et al. [24], the authors examine the data freshness from a functional chain perspective. Accordingly, considering for a functional chain $\tau_1 \rightarrow \cdots \rightarrow \tau_n$ comprised of n tasks such that the data samples produced by the jobs of τ_1 propagate until τ_n. At any time t, the freshness is the difference $t - t_1$, where t_1 is the reading date of the value of τ_1 used to compute the current value of τ_n. It has the meaning of the `last to last delay` as defined in [36]. In [37, 38] the freshness property is used to express the timing gap between the extraction of data from the source and its delivery to the user. In this context, freshness has the meaning of

`currency` [39]. On the other hand, data freshness has the meaning of `timeliness` [40, 41] when expressing the age of the data at the reading time instant by a job of a given task. In this context, the age of the data defines the amount of time elapsed between the data writing and the data reading time instants. In this thesis, the meaning of the freshness is closer to `timeliness`. Precisely, for a pair of tasks τ_i and τ_j such that $\tau_i \rightarrow \tau_j$, for two data samples produced by τ_i at the time instants t_1 and t_2, respectively, the freshness of the data sample produced at t_1 is maintained if the reading of the a job of τ_2 happens at a time instant t such that $t_1 \leq t < t_2$. Also, for the data produced at t_1 time instant, the interval $[t_1, t_2)$ is its validity interval. For `task-to-tasks` category of properties, we consider that `the most recently produced data is always the best`.

Regarding the data consistency property, in Hamann and al. [29] the authors categorize this property into two dimensions: `consistency in value` and `consistency with other variables`. The first dimension requires that the value of a variable should not be affected by action outside the current execution context. The second dimension considers that multiple variables are only valid if all are stemming from the correct (same) point in time. The first dimension represents the `integrity/fullness` of the data. In this context, preserving data consistency means protecting the structure of the shared resource against corruption or modification when the data is still being used by at least one consumer. As for the second dimension, the data consistency property has the meaning of `coherence` between the values of variables in the shared resource structure. Subsequently, for each of member of the shared resource, it must be ensured that only the values produced during the same execution step of the producer task job are consumed.

Example 1 (`Data consistency illustrating example`) *We consider the example taken from [42] to motivate the necessity of ensuring each of the aforementioned dimensions of the data consistency property. In this example, two tasks τ_w and τ_d cooperate to track the coordinate of a moving object and a point is plotted on a screen to display the object trajectory. Precisely, τ_w gets the object coordinates from a sensor and writes them into a shared buffer r where r is a data structure having as members the variable x to store the abscissa and y to store the ordinate of the object in the space. The task τ_d reads the object coordinate from r and plots a corresponding point on the screen, as shown in Figure 1.7.*

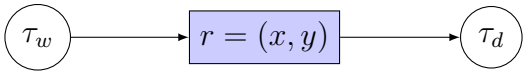

Figure 1.7: *Data consistency motivating example*

In this example it is considered that τ_d has a higher priority over τ_w and the system is scheduled following a preemptive scheduling policy. In Figures 1.8 & 1.9, r is not mutual exclusive, meaning that any released task can access the resource freely any time while in Figure 1.10 r is mutual exclusive; the resource is blocked to other tasks as long as the first task having started the execution of the critical section hasn't yet completed. The gray box corresponds to the task computation time while the blue and the yellow boxes correspond to the reading from/writing into the variables x and y, respectively. The initial position of the object is assumed to be `(1,2)` and, at the execution completion of the first job of τ_w, the object is assumed to have moved to the position `(4,8)`. At the end of the gray box this new position has been already sensed (remained only to written into x and y) and the object has already moved. That to say that, at the execution completion of the job of τ_d, the correct coordinate to be displayed is `(4,8)`. The scheduling results presented in Figures 1.8 and 1.9 show how each of the data consistency dimension may be violated when the shared resource is access free; that is, each of the task access it on its own pace based on its scheduling priority.

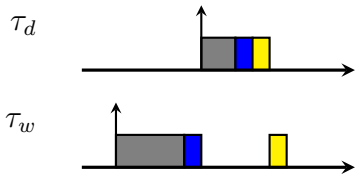

Figure 1.8: *Violation of consistency with other variables example*

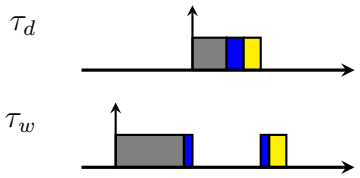

Figure 1.9: *Violation of consistency in value example*

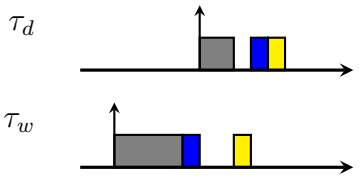

Figure 1.10: *Maintaining data consistency using mutual exclusive resource*

In Figure 1.8 (r=(4,2)), τ_d preempts τ_w after x is updated and before y gets updated. Further, τ_d executes until its completion and displays the position of the object which is the combination of the new value of x and the old value of y; $(4, 2)$. Here the value of x and y are not coherent with respect the exact position of the object. On the other hand, in Figure 1.9 we present the case where τ_d preempts τ_w while still updating x. In this case, τ_d will use the corrupted value of x and the initial value of y, r=(?,2).

Finally, in Figure 1.10 with $r = (4, 8)$ we present the situation where both the data

integrity/fullness and the coherence between variables are ensured by using the mutual exclusive r. Herein, τ_d preempts τ_w after τ_w has entered the critical section and has already updated x which has the value 4. By the time τ_d requests the access to r to read and the point is plotted on the screen it finds that the resource is still blocked by τ_w. τ_w resumes and updates the value of y. In result, at the completion of τ_d, it displays the correct coordinate of the object. Although this ensures the consistency of the data, it may lead to undesirable scheduling situations such as priority inversion, deadlocks formation, chained blocking,$\cdots$. In other words, this solution must be avoided for its negative impacts on the system schedulability. In the scope of this thesis we aim to propose solutions ensuring the data consistency in both dimensions when the shared resource is access free (asynchronous access).

1.4.2.1 Lock-based mechanisms

A mechanism is "lock-based" if and only if a task, accessing a shared resource locked by another task, is blocked. When the lock is released, the task is restored in the ready state and can access the resource. Protocols falling into this category of mechanisms are mostly based on the protocols used in uniprocessor systems. The most popular and accepted of these protocols in the literature include the Priority Inheritance Protocol (PIP) [16], Priority Ceiling Protocol (PCP) [16] and Stack Resource Policy (SRP) [43]. The PIP avoids endless priority inversion by changing the priority of tasks that provoke blocking. Specifically, when a task τ_i blocks one or more higher-priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. By doing so, it prevents medium-priority jobs from preempting τ_i and prolonging the blocking time of higher-priority jobs. Although the PIP bounds the priority inversion phenomenon, the blocking duration for a task can still be substantial because a chain of blocking can be formed. Another problem is that the protocol does not prevent deadlocks. The Priority Ceiling Protocol is used to bound the priority inversion phenomenon and prevents the formation of deadlocks and chained blocking. The basic idea of this method is to extend the Priority Inheritance Protocol with a rule for granting a lock request on a free semaphore. To avoid multiple blockings, this rule does not allow a task to enter a critical section if there are locked semaphores that could block it. This means that once a task enters its first critical section, it can never be blocked by lower priority tasks until

its completion. For that purpose, each semaphore is assigned a priority ceiling equal to the highest priority of the tasks that can lock it. Then, a task τ_i is allowed to enter a critical section only if its priority is higher than all priority ceilings of the semaphores currently locked by tasks other than τ_i. The Stack Resource Policy extends the Priority Ceiling Protocol in three essential points: it allows the use of multi-unit resources, it supports dynamic priority scheduling and allows the sharing of run-time stack-based resources. From a scheduling point of view, the essential difference between the PCP and the SRP is on the time at which a task is blocked. Whereas under the PCP a task is blocked at the time it makes its first resource request, under the SRP a task is blocked at the time it attempts to preempt. This early blocking slightly reduces concurrency but saves unnecessary context switches, simplifies the implementation of the protocol, and allows the sharing of run-time stack resources [42]. Regarding the multi-processor platforms, the last two mechanisms are extended to the multiprocessor priority ceiling protocol (MPCP) [44] and the multiprocessor stack resource policy (MSRP) [45] and they are the most popular for guaranteeing a predictable worst-case blocking time. A significant difference between MSRP and MPCP is that, when a task is blocked on a global resource under MSRP, it busy waits and is not preemptable. This behavior is referred to as a `spin-lock`. A FIFO queue is again used to grant access to tasks waiting on a global resource when it is unlocked. MSRP provides both a bounded blocking time and bounded increases in task execution times due to the spin-locks. The flexible multiprocessor locking protocol (FMLP) [46] has been proposed. It is deemed to manage short resource requests using a busy-wait mechanism (as in MSRP) and long resource requests using a suspension approach (as in MPCP). Parallel-PCP is proposed in [47] for tasks scheduled using a global fixed-priority preemptive algorithm. The multiprocessor hierarchical synchronization protocol (MHSP) [48] is used to manage resource sharing for both partitioned and global scheduling approaches.

1.4.2.2 Lock-free mechanisms

The Lock-free mechanisms appear as an alternative to lock-based mechanisms. Specifically, lock-free mechanisms allow each reader to access the communication data without blocking. A lock-free algorithms is proposed by Anderson et al. in [49] which are similar to those

used in optimistic concurrency control in database systems. They allow immediate access to the resource, but latter check to see if there was a conflict over this access. If there was, then computations are abandoned and the resource is re-accessed. These algorithms are therefore lock-free, but can involve looping while the task is waiting to gain conflict-free access. To preserve the data consistency property, the μORB communication mechanism [50] implements a lock-free mechanism. The issue regarding the bounding of the number of retries (loop) is addressed in [49, 51]. Both mechanisms (lock-based and lock-free) have in common that they are susceptible to causing some blocking of scheduling depending on the working conditions. For lock-based, this would come from the blocking of the shared resource while there is a task which has already entered the critical section. For lock-free, if the communication is causally dependent, the one considered in [24, 52] for instance, where, for a task τ_i to start its execution, a task τ_j must have already produced an output (dependent tasks). Thus, if τ_i is late to produce the output (due to an eventual high number of retries) then τ_j has to wait, and if the same happens repeatedly for the output of τ_j, this behavior may turn into a blocking chain. In consequence, the use of lock-free mechanisms in both hard real-time systems and model-based design is unsuitable, because of the large worst-case blocking time limit and the high time penalty induced by the need to repeat the operation in case of concurrent access. Therefore, in comparison to other mechanisms, there has been only few results on the use of lock-free mechanisms for hard real-time systems since the lat '90s [53].

1.4.2.3 Wait-free mechanisms

Wait-free mechanisms consider the shared resource to be a communication buffer capable of holding more than one copy of data such that access to these data is possible with no waiting (asynchronously) time while avoiding possible corruption of such data. Precisely, the data producer and consumers are protected against concurrent access by replicating the communication buffer and by leveraging scheduling information such as tasks temporal characteristics, priority and scheduling priority [54, 55]. In wait-free mechanisms, the only shared resource is the buffer index, which can be updated atomically or with a short critical section. This way, these mechanisms avoid possible blocking of resources on writing or reading

the data. Each job (whether of the producer task or of the consumer task) accesses the buffers at its own pace based on its priority. In fact, it is ensured that every time a producer needs to update the communication data, there is always a booked unused buffer slot to write in. In other words, all the jobs having read from this slot will have completed their executions. The solutions proposed in the literature falling into this category vary mainly in terms of the buffer access policy and the required size per proposed solution. In [54] the authors introduce a single-writer, multiple-reader wait-free algorithm using the `Compare-and-Swap (CAS)` instruction which takes three operands as follows `Compare-and-Swap (mem,` v_1`,` v_2`)` where the value v_2 is written into the memory location `mem` only if the current value of `mem` is equal to v_1. Otherwise, the value at `mem` is left unchanged. It uses a global variable, *Latest*, that indexes to the most recently written message buffer. Additionally, each reader has an entry in a usage array indicating the buffer it is using. When the reader reads, it first clears its entry, and then uses `CAS` to atomically set this to *Latest* if it is still cleared. It then reads back the value from its entry, and can then safely read from the indicated buffer. Chen's algorithm requires $p + 2$ message buffers, where p is the number of reader tasks. In [55], the authors propose three algorithms intended to improve the Chen's wait-Free algorithms proposed in [56], namely, the `Improved Chen's Algorithm`, `Double Buffer algorithm` and `Improved Double Buffer Algorithm`. These algorithms have the vocation of taking advantage of the real-time properties of the communicating tasks to reduce both the time and space overheads of class of `wait-free single writer, multiple-reader` inter-process communication algorithms. According to the authors, at that time, the existing wait free asynchronous algorithms have higher space overheads with a significant execution overheads than the synchronization-based algorithms such the ones proposed in [56–59].

With consideration of the real-time properties of communicating tasks, the improved Chen algorithm partitions the set of readers into two sets: `the fast readers` and `the slow readers`. By doing so, this algorithm reduces the space required from `p+2` to `m + max(2; n)` where `m` is the number of slow readers and `n` is the number of buffers required by the fast readers. Given that `n` is chosen to be less than or equal to the number of fast readers (i.e. $n \leq p - m$), the improved algorithm requires no more buffer space than the original algorithm. In the worst-case (i.e., all readers are slow readers), the improved algorithm simply

degenerates into the original algorithm. Also, the run-time overheads will be greatly reduced, since the fast readers use the non-blocking writer mechanism [58] and the writer overhead is linear to the number of slow readers only, rather than to all readers. Therefore, space and time overheads can be reduced. In general, regarding the computation of the required size of the buffer, two methods are mainly used, namely, the `reader instance` [54] and the `lifetime bound` [54, 58] methods. The former relies on the computation of an upper bound for the maximum number of buffers that can be used at any given time by reader tasks while the latter is based on the computation of an upper bound on the number of times the writer can produce new values while a given data item is used by at least one reader. The reader instance method requires a buffer size equal to the maximum number of read task instances that can be active at any one time (the number of read tasks if the delays of the delays are not greater than the periods), plus two additional buffer. When all readers have a lower priority than the writer, then only `n+1` buffers are needed. This is taken into account by the `Dynamic Buffering Protocol, DBP`, which ensures the data consistency with an execution complexity of `O(n)`. On the other hand, regarding the `lifetime bound method`, it is considered by the so-called `Temporal Concurrency Control Protocol (TCCP)`. The corresponding run-time complexity is evaluated to `O(1)` while the number of required buffers depends on the lifetime of the data.

Notice 5 *In this thesis we consider the* ***`wait-free mechanism`*** *and the computation of the required buffer sizes is based on the* ***`lifetime bound method`*** *especially for its lower run-time complexity.*

1.4.2.4 Read-Execute-Write mechanism

This mechanism is based on the `read-execute-write` semantic. According to this semantic, each reader task makes a copy of each of its input resources which are used throughout the execution including when the the task resumes after being preempted by higher priority tasks. At the execution completion, the output data are written back to the corresponding shared variable. So, instead of allowing an explicit (direct) access to the communication resource, this mechanism envisages an implicit (indirect) access. The communication models resulting from these data access strategies are referred to as `explicit` and `implicit` communication models.

For `explicit communication models` the data consistency is ensured either by setting the shared resource mutual exclusive and then applying the `lock-based mechanisms` or setting the shared resource non-mutual exclusive and using the `lock-free mechanisms`. Herein, the executing task is allowed to access directly the shared resource to read/write the values anytime a read or write operation to the resource is demanded. This results in uncertainty, since the exact point in time the resource access is performed depends on the respective execution path of the task [60]. An example of an explicit inter-tasks communication model is presented in Figure 1.11 where 3 tasks τ_1, τ_2, and τ_3, communicate through the shared resource r_1 such that τ_1 is the producer, τ_2 and τ_3 are the consumers. All the tasks are assumed to have implicit deadlines, scheduled using the task-level fixed priority policy. The followings are their timing parameters values: $\tau_1(1,3)$, $\tau_2(1,5)$ and $\tau_3(5,15)$; the first parameter is the worst-case execution time while the second is the task period. The notations $fr(t_1,t_2,\cdots)$ where t_1 and t_2 are the time instants such that $t_1 < t_2$, correspond to the different activation time instants (the first activation). Analogically, since we consider a preemptive scheduling algorithm, $rs(t_1,t_2,\cdots)$ corresponds to the time instants where the jobs of a given task resume from being preempted by the ones of the higher priority tasks. The same applies to the writing time instants of the producer tasks denoted by $wr(t_1,t_2,\cdots)$. To preserve data consistency, τ_1 must be denied permission to write into r_1 if there is a job of τ_2 or τ_3 currently executing, otherwise, it may cause data overlapping.

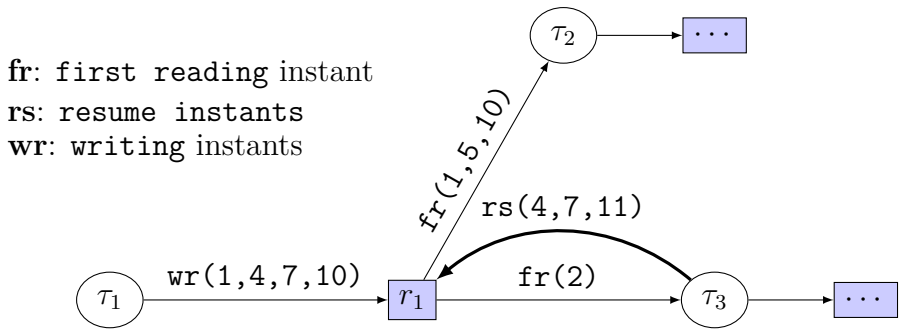

Figure 1.11: *Example of an explicit access to the shared register.*

Data overlapping occurs if, by the time a job of the consumer resumes and needs to read the data, it finds that the producer has already overwritten the data initially used. The lock-free mechanisms address this problem but may cause blocking with the possibility of

disturbing the scheduling of tasks. For a single size shared resource, an other alternative avoiding any interference to the tasks scheduling is the implicit communication model. The implicit communication model is ensured in a way that, before using a data, each job of the consumer task must first copy the data from the global variable into a location accessible for this job only (a local copy of the variable). Further, during the execution run-time this job uses the data from the local copy. In Figure 1.12 we depict the stages involved in the implicit communication. In this figure we have 3 tasks τ_1, τ_2 and τ_3 communicating via a shared resource r_1. The figure is divided into two parts by the horizontal axis. The part below the axis shows what happens at the activation and completion times for a job of a consumer task. The upper part of the figure shows the handling of the copied data during the job run-time interval. So, considering the shared global resource r_1, at the very first activation, each job of its jobs it makes a copy (cIn_i) of the data being into the shared global variable r_1. During the execution time, including resumes from preemptions, this job will continue to use the copied data. So, even if the τ_1 generates new data, it simply overwrites the value within the global variable r_1. At the execution completion, the output result written into the $cOut_i$ is copied back to the corresponding shared global variable. The `implicit communication model` is implemented in AUTOSAR [7].

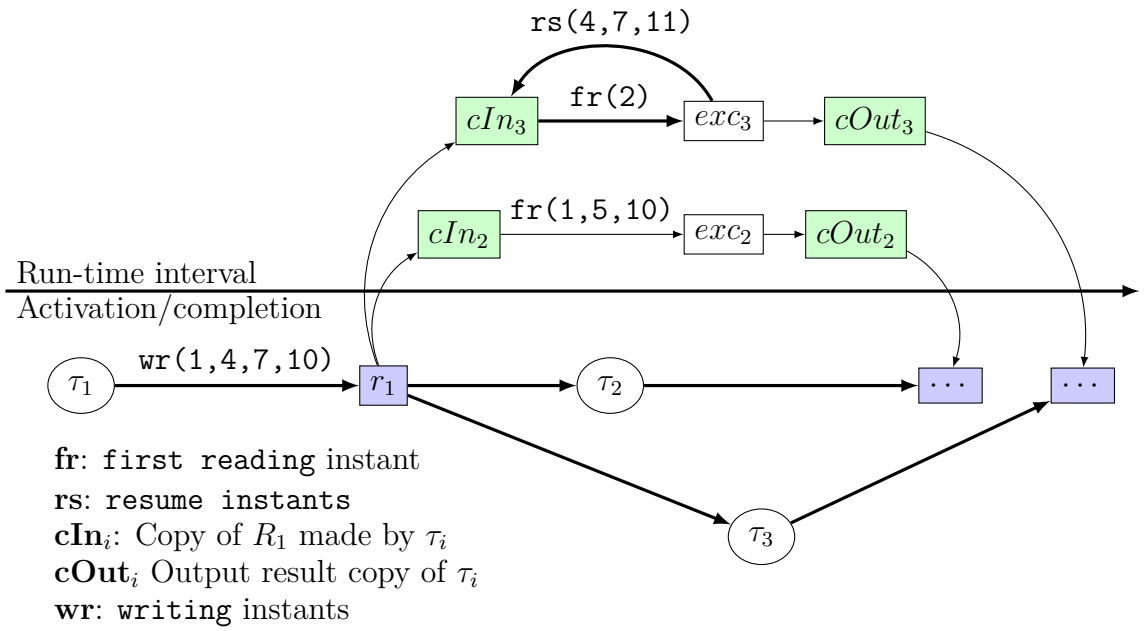

Figure 1.12: *Example of an implicit access to the shared register.*

Although guaranteeing data consistency, this communication model has some drawbacks such as dynamic memory allocation of local variables that is not predictable but also the significant access latency which dependent on multiple factors including, the cost of access to remote and local memory, number of accesses to the label during one execution to the local memory, and the period/activation rate of the task. However on the memory storage front, more local storage is required, since for every task which accesses the label, an extra local copy is required [29]. This result in long *buffering segments* [61].

Into this category falls also the Logical Execution Time (LET) [62] paradigm. LET considers that the reading of the data is done at the job release time whereas the generated output data is available only at the end of the period. Precisely, a data produced during the k^{th} period can only be read by a job activated at the beginning of the $(k+1)^{th}$ period. LET paradigm stems from the Giotto [63] programming formalism. This communication model is deemed to be predictable provided that the writing and the reading instants happen at predictable instants (at the beginning of the period and at the end of the period). That being, LET suffers from the largest propagation delays [30, 64] and, the same way as the implicit communication model [7], LET requires local copies to temporally store the produced output data until the next period.

1.4.3 End-to-end data properties

The data properties required to be maintained at the level of the functional chain, also referred to as `end-to-end properties`, determine the amount of time required for a data to propagate from the first task until the last one with a positive (correct) impact on the system functioning. This amount of time, also referred to as end-to-end propagation delay, must be within a bounded time interval. Over the last decades the verification of the end-to-end data properties constraints has been the subject of several results in the literature, where authors are interested in computing the worst-case, the best-case or the average end-to-end propagation delays for a given functional chain. In [36] the authors propose a framework distinguishing different meanings of end-to-end timing depending on the system requirements. For instance, control engineers are mostly concerned with the *maximum age of data* which is the worst-case timing of the latest possible signal while in the body electronics, the *first*

reaction is key, i.e., the worst-case timing of the earliest possible signal. Herein, four different end-to-end timing semantics to characterize the timing delays of effects in the context of multi-rate tasks communicating through shared register are described and computed. Generally, the end-to-end properties regarding a single functional chain are verified on the basis of different end-to-end delay metrics such as *age latency* and *reaction latency*.

On the other hand, verifying end-to-end data properties goes beyond the consideration of a single functional chain by examining the end-to-end properties of data propagating through different functional chains. As an example, in [27] the authors propose two configurations of functional chains, namely the `divergent` and `convergent` functional chains.

An example of the divergent functional chains is shown in Figure 1.13 while an example of the converging functional chains is shown in Figure 1.14.

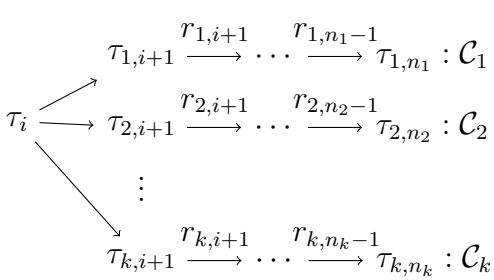

Figure 1.13: *Example of k diverging functional chains.*

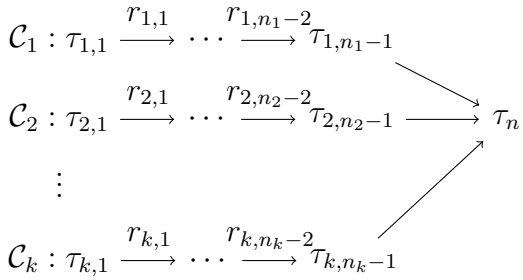

Figure 1.14: *Example of k converging functional chains.*

From the example presented in Figure 1.14, a converging task τ_n reads the data propagating through k functional chains where each of the functional chains has a different source task. Each functional $\mathcal{C}_j,\ \forall 2 \leq j \leq k$ is composed of n_j tasks. Reversely, from the example presented in Figure 1.13, the data produced by the task τ_i propagate from τ_i until the last task $\tau_{j,n_j},\ \forall 2 \leq j \leq k$ where n_j is the number of tasks composing the functional chain $\mathcal{C}_j$. In both examples, $r_{j,i}$ is the resource shared between the tasks τ_i and τ_{i+1} both belonging to the same functional chain $\mathcal{C}_j,\ \forall j \leq k$.

Regarding the verification of different end-to-end constraints for the convergent and divergent functional chains, authors in [25] addressed the latency and freshness analysis applied to the Integrated Modular Avionics (IMA) architectures. Herein, the latency corresponds to the time elapsed between an event at the beginning of a functional chain and the first event

depending on it at the end of the chain while freshness is looked at as the time between an event at the end of a functional chain and the earliest dependent event at the beginning of the chain. In [26], the authors address the worst-case temporal consistency in Integrated Modular Avionics Systems. A temporal consistency constraint verification is required between divergent functional chains sharing at least a data at their beginning. A temporal consistency requirement between convergent functional chains is expressed on chains sharing at least a data (and the function processing the data) at their end. Although the authors address the problem related to the verification of the aforementioned end-to-end data properties constraints, it follows that each of the verification of theses properties for convergent and divergent functional chains is dealt with in isolation.

Moreover, in the practice, both the convergent and divergent functional chains may be met in one configuration that we referred to as the `spindle propagation chains`. An example of such a configuration is presented in Figure 1.15. For such a configuration, the data to be used by the converging task should be originally produced by the diverging task (with reference to convergent and divergent functional chains).

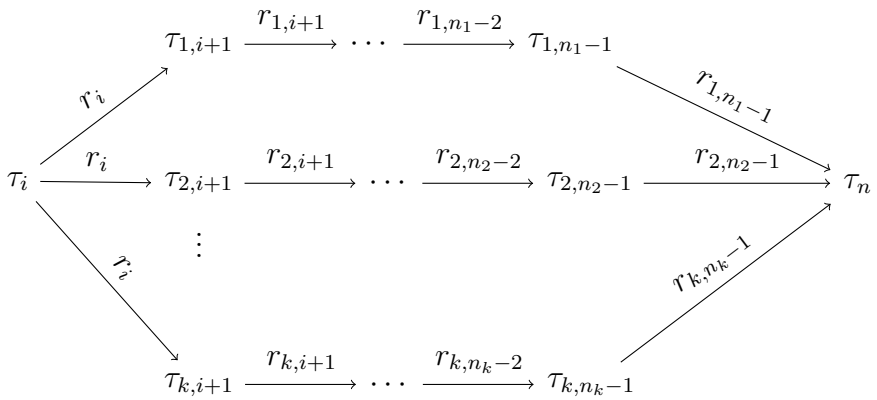

Figure 1.15: *Example of the* `spindle propagation chains`

Regarding a spindle, the goal of this thesis is to guarantee that the spindle fine task will associate only data resulting from the same execution step of the spindle source task. This property consisting in associating only the data resulting from the same execution step of the producer is referred to as `data matching property`. To ensure such a property, we first need to compute the end-to-end latency and the sizes of the waiting buffers at the end of each chain. In the real-time literature several solutions regarding the end-to-end latency

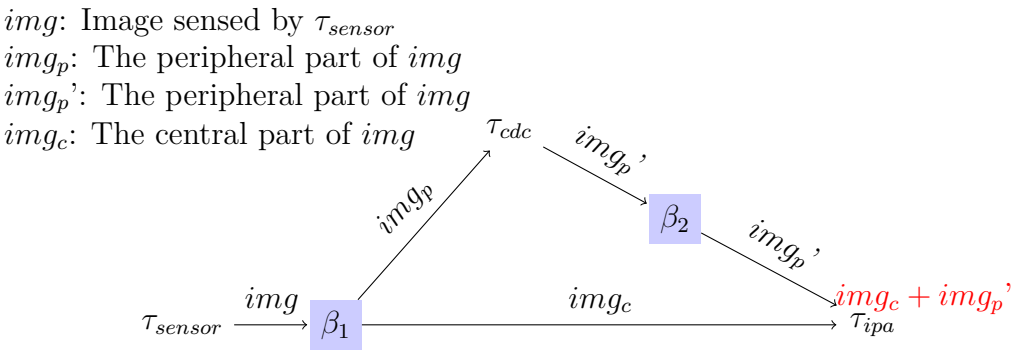

Figure 1.16: *Data matching motivating example.*

computation exist such as those presented in [25]. As for the data matching issue, this problem has been addressed by Pontesso and al. in [65, 66] in a context different from that of this thesis while considering different assumptions.

To illustrate the need to guarantee this property, we present below the example of the FADE system [67] which is considered in RTMaps tools [68].

Example 2 (`Data matching property example`) *FADE is a vehicle detection and tracking system composed of a set of image processing components in charge of detecting the characteristics (detection of shadows, headlights, etc.) related to the presence of a vehicle in the neighborhood (Figure 1.16).*

The system is composed of the `image acquisition component`, the `crop and decimate component` and the `image processing component` denoted, respectively by τ_{sensor}, τ_{cdc} and τ_{ipa}, respectively.

The task τ_{sensor} captures the image of the object in the neighbourhood in high resolution. The central part of the image is more clear (img_c) and easily identifiable while the periphery of the image (img_p) not very clear. In order to increase the object identification accuracy, the task τ_{cdc} gets the img_p as input data, performs a deep analysis on it and the output result, denoted by imp_p' is written into the buffer β_2. Finally, the task τ_{ipa} combines img_c and img_p' and infers the identity of the crossing object.

Obviously, the associated parts of the image are shifted by a certain delay induced by the processing of the τ_{cdc} depending on the sampling periods of all the tasks involved in this identification. In order to avoid potential performance degradation, the τ_{ipa} instances must

read the data resulting from the same execution step of the τ_{sensor}. Otherwise this may lead to incorrect results. However, the challenge here is twofold:

- the system of tasks may be multi-rate; that is, the communicating tasks have different periods which may leads the lost of some unused data. That being, it follows that the concerned functional chains may have different propagation delays. Assuming that the propagation delay of $\tau_{sensor} \xrightarrow{\beta_1} \tau_{cdc} \xrightarrow{\beta_2} \tau_{ipa}$ is always that the one of $\tau_{sensor} \xrightarrow{\beta_1} \tau_{ipa}$, at the release of a job of τ_{ipa} it may happen that the corresponding part of img_p' is already overwritten from β_1. So, what should be the size of β_1 and the data access strategy guaranteeing that τ_{ipa} always finds the corresponding parts of the image and of-course retrieves the ones resulting from the same execution step of τ_{sensor}.

- if the system of tasks is scheduled preemptively, how to guarantee that none of the combined data is corrupted as shown in Figure 1.9. Data corruption here means combining the old and the new parts of the data (data overlapping). Of-course this is an undesirable case as it might lead to erroneous results or performance degradation.

Chapter

2 Unmanned aerial vehicles (UAVs)

Chapter content

2.1 Related work 33

2.1.1 Communication and control system in UAVs 35

2.1.2 Coupling effect in UAVs 37

2.2 The PX4 autopilot 39

2.2.1 The Flight Stack 40

2.2.2 The middleware architecture 42

2.3 The thesis context 51

2.3.1 The PX4-RT system 52

2.3.2 The thesis goals 53

2.3.3 Identifying data constraints covered in this thesis 53

2.1 Related work

Unmanned aerial vehicles (UAVs) are an important tool for both military and civil areas. These machines are characterised by their agility, versatility, low cost and easy-to-deploy properties. According to their usage, UAVs can vary in size, shape, computational power, number of sensors and cameras, speed and maneuverability. Whatever the features they have, each UAV can be interpreted as a cyber-physical system (CPS) which executes the internal loop consisting of initial data acquisition, information exchanging, decision making

and the final execution. CPS achieves the tight coupling between the cyber domain and the physical domain by strictly embedding the cyber processes into the physical devices. Thus, the reliable, real-time and efficient monitoring, coordination and control to the physical entities can be conducted through the closed loop. For example, in the UAV networks, the data sensed by the sensors originate from the physical world (i.e., its mission circumstance), and the final decisions, made by computation and conveyed by communication, are translated into instructions and eventually take effects on the physical world through the actuators.

In this chapter we present related work with respect to the contributions on UAV as cyber-physical systems while concentrating on the communication and network components.

Nowadays, the use of CPSs evolved from military to almost every domain of our life, like transportation, energy, healthcare and manufacturing [69]. The principle of CPS is to monitor and control the physical world by integrating the cyber processes, including sensing, communication, computation and control, into physical devices. All these functionalities can be seen as an extension of control systems and embedded systems [70].

The belonging of UAVs to the cyber world can be proven by the UAV's features and functionalities. Usually, an UAV has an on board unit responsible for computation, one or multiple sensors and actuators used mainly for navigation, and a network capable of transmitting data between the others components of the UAV or towards exterior devices. For a better formalisation, we present in the following the main components of an UAV:

- the hardware, belonging to the physical domain, which directly interacts with the cyber and physical world by sensing and actuating, or provide computation and communication capabilities. It may include the sensors, actuators, computation chips, communication equipment and so on;
- the software, belonging to the cyber domain, which is used to manipulate the hardware, as well as analyze the data and make decisions. It may include the embedded operating systems, application programs, functional algorithms and so on;
- the wireless communication network, belonging to the cyber domain, which is responsible for exchanging and sharing information among the entities in the system. It may include the communication modules, standards and protocols (e.g.,the medium access control (MAC) and routing protocol);

- the cloud service platform, belonging to the cyber domain, which is a highly integrated, open and shared data service platform. It can be a cross-system, cross-platform and cross-domain information center which incorporates data distribution, storage, analysis and sharing.

2.1.1 Communication and control system in UAVs

From the CPS perspective, an UAV network is an integration of sensing, communication, computation and control. In detail, sensing introduces the original data in to the UAV network from the physical world. Communication drives the data to flow inside of the UAV network, which guarantees the information distribution and sharing, and thus a global analyzing and deciding. Computation is the key of analyzing and decision making based on all the acquired information, while **control** focuses on translating the decisions into instructions through the actuators acting on the real world finally. We observe that the communication is the link between the other three elements of the system while the control is the one allowing the UAV to function properly. Communication and control play a major role in safe and efficient coordination of the UAV.

We can divide the external communication in two categories: (i) UAV to UAV communication and (ii) GCS (ground control system). For both of these communication types, different data are exchanged and according to the data type, in [71] the author presents a classification of three types: control traffic, coordination traffic and sensed traffic. The control traffic exchange enables the GCS to monitor and influence the behaviors of the UAVs. It includes the mission commands and flight control messages from the GCS to UAVs and the status data of the UAVs (e.g., the health status and telemetry data,including the inertial measurement unit (IMU) and global position system (GPS) information). The coordination traffic means any data that needs to be exchanged for local decision making, cooperation and collision avoidance, without explicit input from the GCS. This kind of traffic may include the telemetry data, way point, mission plan and so on. The sensed traffic encompasses the on board sensor data used to measure the physical environment, which is transmitted to the GCS considering that on board analysis of the sensor data may not be reasonable. It includes data transmission of various size (from weather sensor readings to high-quality images

and videos) for real-time monitoring on the front line, and decision making or post-mission analysis in the GCS.

In the survey papers [72] and [73] the authors underline the communication challenges caused by the high mobility and energy constraints of UAVs. Some examples of such challenges are: intermittent links, fluid topology, Doppler effect, complicated antenna alignment and vanishing nodes.

The **control** is responsible for the precise execution, which is embodied in a series of actions that affect the UAVs themselves and the physical world. The closed-loop data flow ends at control, and the previous sensing, communication and computation make sense only when the decisions are translated into instructions on the actuators and finally make a difference. In the rest of this section, we care more about the flight control of the UAVs, which is the precondition of the task execution.

The control to the UAVs varies with the UAV forms, such as the fixed-wing, multirotor, monorotor/single-rotor, airship and flapping wing. They would suffer different aerodynamics, and are designated with different missions according to their characteristics as mentioned in [74, 75]. For the specific missions, the control may include the macro flight control decisions (e.g., the path planning and formation control) and the micro executions of the actuators (e.g., the motors and rotors) according to the generated instructions.

The **flight controller**, also known as autopilot system, is the core component of an UAV. It is normally used to realize the autonomous flight control, including the attitude stabilization, fight waypoint generation, mission planning and so on [76]. In order to achieve these functions, the flight controllers need the hardware and software supports concurrently. The former mainly contains the on board computers,inertial measurement units, various sensors, GPS modules, communication devices, power management modules and so on [77], and the latter usually includes the task allocation, path planning, waypoint generation, attitude control and signal processing algorithms [78]. The flight controllers enable the UAVs to develop from the simple remote-controlled aircrafts to the fully autonomous and intelligent aircrafts.

Some examples of open-source flight controllers are: Paparazzi, PIXHAWK, Phenix Pro, OcPoC, DJI A2, NAVIO2 and Trinity. These controllers are reviewed in [76, 79, 80], while the PIXHAWK is detailed in this document being the chosen platform for the experimental

part of the thesis.

In order to guaranty a stable, smooth and safe flight from take off to landing, the control relies on **flight control algorithms**. Depending on the type of UAV and the mission supposed to accomplish, these algorithms vary in functionality, size and reliability. In [81–83], the authors survey the existing algorithms and present the main challenges that they encounter (e.g. static instability, need of fast control response) for both linear and non-linear approaches.

2.1.2 Coupling effect in UAVs

In the UAV networks, with a data flow-based closed loop being built, the cyber-domain components and the physical-domain components are tightly coupled. In this section we present the three types of coupling: computation-communication, computation-control and communication-control. For each of these relations we describe the main influences as well as the existing literature improving the UAV's performance relying on these couplings.

2.1.2.1 Computation and communication

Computation and communication are promoting each other through the fact that communication contributes to computation while computation boosts communication.

The impact of communication on computation is straight forward by the fact that the input and output of computation are obtained and announced through communication flows. In a hard real-time system, the performance of communication has a high impact on computation, with tasks that need to finish execution between a certain deadline and for which an abnormal jitter can hinder the success of the task. Moreover, in a large scale network consisting of UAVs with high computation power, the cooperative computing and the swarm intelligence could be boosted by the communication network among the individuals.

Moreover, computation enhances communication, and is expected to unlock, at least approach, the communication bound. There are many researches focusing on embedding the computation into communication to improve the communication and the networking performance for the UAVs, in other words, exchanging for communication using the computation. They can be classified into two categories, which adopt quite different methods. The first

category enhances the performance of the exact communication and networking themselves, from different layers of the network. Some intelligent algorithms are adopted to cope with the challenges, such as low latency and high transmission rate requirements, which are brought by the intrinsic features of the UAVs (e.g., the frequent topology changes and status interactions, and also the high-resolution image or video transmissions). In the second category, the results concern on lessening the communication quantity and overhead in the whole system, considering the cost of communication. The agent decision technology is introduced to let UAVs decide whether to, when to, what to, and whom to communicate [84–88].

2.1.2.2 Computation and control

In the UAV networks, control highly depends on computation since the control decision making are usually undertaken by the on board computer, and the computation outputs, are directly translated into continuous signals and fed to the actuators. One method to boost the computation and, subsequently the control, is for the UAVs to be equipped with performant hardware (e.g. high-performance processors, fast and large memory).

Another possible approaches decreasing the complexity of the nonlinear control design is to adopt learning algorithms, allowing the training of suitable control actions. Such algorithms that enhance the flight control through computation belong (and are not limited) to classes of algorithms like fuzzy logic-based flight control algorithms [89–91], artificial neural network-based flight control algorithms [92, 93] and reinforcement learning-based flight control algorithms [93–95].

The enhancement of control for UAVs in formation can be achieved through bio-inspired algorithms [96] or through artificial intelligence algorithms [97].

2.1.2.3 Communication and control

Communication and control are also two tightly coupled components, and they constrain and promote each other. As the intrinsic feature of the UAVs, the three-dimension mobility is dominated by the flight control, which brings more challenges to the communication and networking. In return, the flight control also creates a novel method to solve the problems in communication by integrating the physical domain, e.g., location changes, other than only

adjusting the communication parameters.

It has been shown that a communication scheme needs to be adopted to increase the aggregated maneuverability of mobile agents [98]. At the same time, the performance of the communication has a significant effect on the flight control of the UAVs. For example, the communication delay may bring the risk of collisions in a UAV swarm [99]. In return, the communication also rely on the flight control. The communication network among the UAVs is guaranteed by the optimal topology control, or at least, the proper distance maintenance between each node, which should be considered when doing the formation control.

Communication and control also promote and benefit from each other. Intuitively, the performance improvement of the communication motivates the formation control by efficiently delivering the status among the UAVs. Thus, dramatically, flight control of the UAVs may benefit a lot from the existing researches which focus on dealing with the communication challenges exactly brought by it [100, 101].

In Section 2.2 we present the PX4 autopilot which is the main component of PIXHAWK control system. We use this autopilot to exemplify the coupling between the control, communication and computation for a precise use case.

2.2 The PX4 autopilot

PX4 is an autopilot developed by world-class developers from industry and academia, and supported by an active world wide community, it powers all kinds of vehicles from racing and cargo drones through to ground vehicles and submersibles [52]. Being an open source autopilot flight stack, it has been considered by the CEOS project partners with the intention of carrying out the critical missions on the highly reliable drones. The main improvement expected at the end of this project is to transform the current version of PX4 into a real-time version that further called PX4-RT. This task is being accomplished within INRIA [1] research center by the KOPERNIC [2] team.

[1] French Institute for Research in Computer Science and Automation

[2] Keeping worst-case reasoning appropriate for different criticalities. The KOPERNIC team deals with the problem of studying time properties (execution time of a program or the schedulability of communicating programs, etc.)

The PX4 structure consists of two main layers: the **flight stack** and the **middleware**. The flight stack is an estimation and flight control system while the middleware is a general robotics layer that can support any type of autonomous robot, providing internal/external communications and hardware integration.

All PX4 air-frames share a single code base, including other robotic systems like boats, rovers, submarines etc. The complete system design is reactive; all functionality is divided into exchangeable and reusable components and communication is done by asynchronous message passing.

The diagram in Figure 2.1 provides a detailed overview of the building blocks of PX4. The top part of the diagram contains the middleware blocks, while the lower section shows the components of the flight stack.

2.2.1 The Flight Stack

The flight stack is a collection of guidance, navigation and control algorithms for autonomous drones. It includes controllers for fixed wing, multirotor and VTOL airframes as well as estimators for attitude and position.

The diagram in Figure 2.2 shows an overview of the building blocks of the flight stack. It contains the full pipeline from sensors, RC input and autonomous flight control (Navigator), down to the motor or servo control (Actuators). We enumerate the roles of the main subsystems:

- An estimator takes one or more sensor inputs, combines them, and computes a vehicle state (for example the attitude from IMU sensor data).
- A controller is a component that takes a setpoint and a measurement or estimated state (process variable) as input. Its goal is to adjust the value of the process variable such that it matches the setpoint. The output is a correction to eventually reach that setpoint. For example the position controller takes position setpoints as inputs, the process variable is the currently estimated position, and the output is an attitude and thrust setpoint that move the vehicle towards the desired position.
- A mixer takes force commands (e.g. turn right) and translates them into individual motor commands, while ensuring that some limits are not exceeded. This transla-

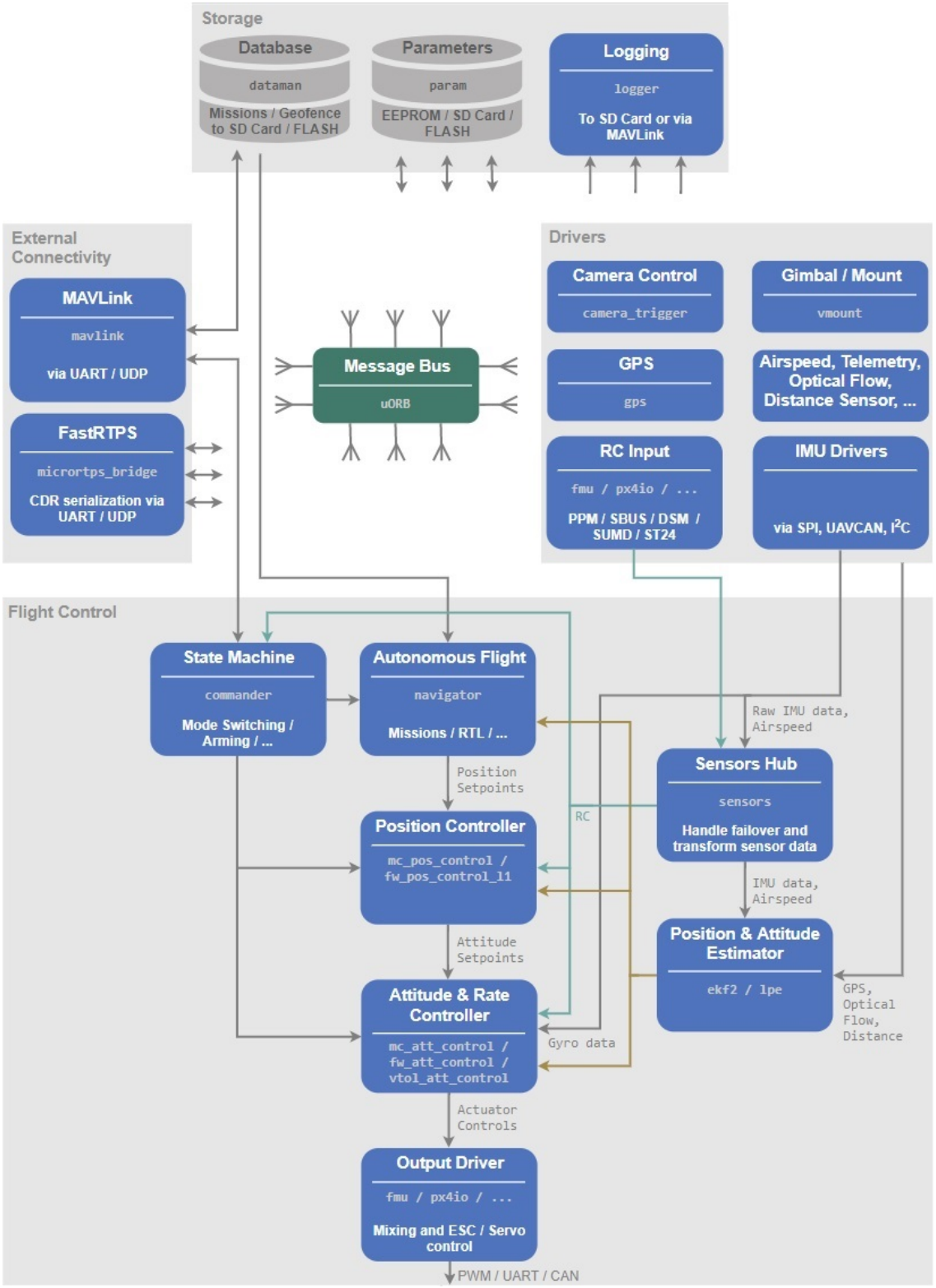

Figure 2.1: *The PX4 High-Level Software Architecture*

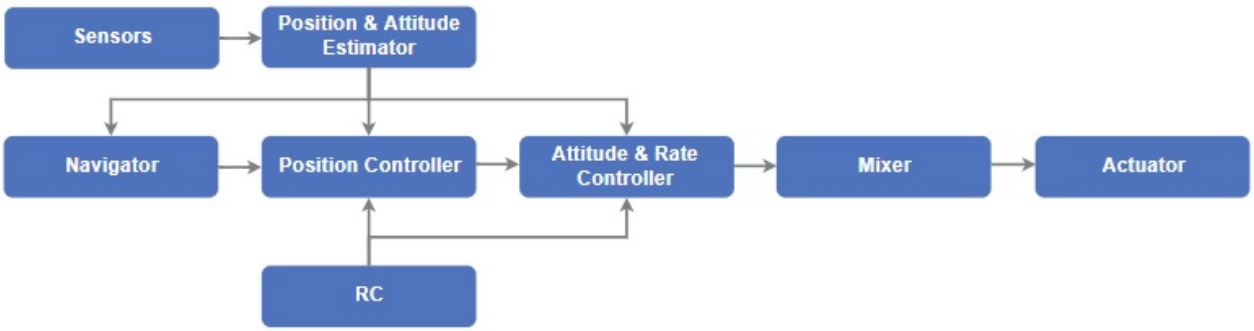

Figure 2.2: *The PX4 flight stack diagram*

tion is specific for a vehicle type and depends on various factors, such as the motor arrangements with respect to the center of gravity, or the vehicle's rotational inertia.

In the following sections we present the diagrams for the main PX4 controllers.

2.2.2 The middleware architecture

In this section, we describe the internal communication middleware architecture of the PX4 system, consisting of the internal and external communication mechanisms. The internal communication mechanism ensures communications between the PX4 modules/tasks while the external communication mechanisms enable communication between the system PX4 systems and the off-board companion computers as well as the Group Control System (GCS). The PX4 internal communications are based on the μORB (micro Object Request Brocker) [50] communication mechanism, while the external communications are handled by the MAVLink or RTPS communication mechanisms. Given that in this thesis we focus on the internal communications, their management within the PX4 system is going to be extensively described in the following section. Regarding the external communications, the readers are invited to refer to [102] for MAVLink and [103] for RTPS for more details.

The μORB is an asynchronous messaging API following following the *publish/subscribe* paradigm [50, 104–107]. Modules communicate by passing or receiving messages where each message is called `topic`. Publish and subscribe communication systems are widely used in distributed systems. In contrast to the traditional point-to-point model such as client-server, the publication/subscription model decouples the publisher and the subscriber in time, space, and synchronization [106]. The producer publishes the topics to which consumers of these

topics will subscribe. Precisely, when the producer publishes new news events related to a specific topic, the server (the event brokerage system) distributes and notifies the consumers of the arrival of the new events. Therefore, each consumer can access the publications asynchronously, anytime and anywhere, at their convenience. The publish/subscribe model is relevant when the date transfer is shared out between of many consumers, events or data updates are not frequent, when events are common interest or when the deadline is shorter. Conversely, the public/sub model is not appropriate when consumers rarely use published data. While the publish/subscribe model is widely used in distributed systems, its implementation is domain-dependent. For instance, Tripakis and al. [107] has implemented it in a real-time system in a completely different manner compared to the one adopted in PX4 [52], which is not subject to real-time constraints. And, of course, this is also true for the costs in terms of memory and CPU overhead. Accordingly, below we describe the implementation of the publish/subscribe model within the PX4 system. This description is made having in mind the calculation of the memory space consumption, the ease of implementation in an embedded real-time system and the cost to be paid to guarantee the data properties deemed necessary for the efficient functioning of the entire system according with respect to this thesis purposes.

2.2.2.1 The μORB topics management

As an example of topic data structure, it is shown in the figure 2.3 where the `follow_target` topic is considered. All topics are stored locally in the directory "`msg/`". This directory contains as many files as there are topics in the system provided that there is one file per topic. Each topic file has the extension `.msg`. The naming of these files follows the following rule: *for a subject named* ***`topic_name`*** *the name of the corresponding file will be* ***`topic_name.msg`***. For example, considering the topic `vehicle_attitude_setpoint`, its corresponding file is named `vehicle_attitude_setpoint.msg`. Besides the list of files per topic, the directory "`msg/`" contains also the file `CMakeLists.txt`. A non-exhaustive list of topic files contained in the "`msg/`" directory is shown in Figure 2.3.

The file `CMakeLists.txt` is the debugging entry. Basically, once the system is debugged, each topic file is converted to a `C/C++` header using the `genmsg_cpp` library. The correspond-

ing headers are stored in the "uORB/topics/" directory. A list of the corresponding C/C++ headers is shown in Figure 2.4.

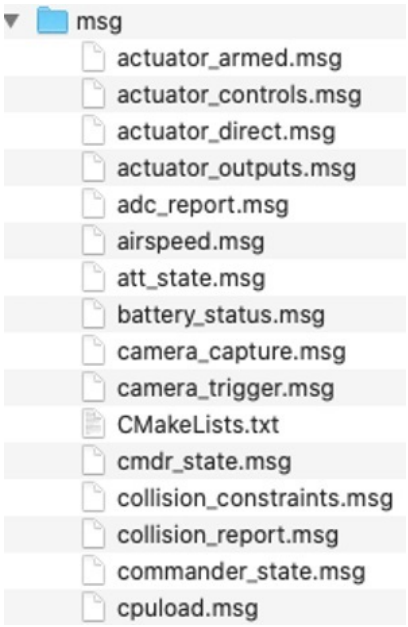

Figure 2.3: *The non-exhaustive list of the µORD topics in the local directory*

```
uORB/topics/
  actuator_armed.h
  actuator_controls.h
  actuator_direct.h
  actuator_outputs.h
  adc_report.h
  att_state.h
  battery_status.h
  camera_trigger.h
  camera_capture.h
  cmdr_state.h
  collision_constraints.h
  collision_report.h
  commander_state.h
  cpuload.h
```

Figure 2.4: *The C/C++ header derived from the topic depicted in Figure 2.3*

The *µORG* communication mechanism is internally based on the `read`, `write` and `ioctl` file descriptors used to manage the subscription to topics.

2.2.2.2 The *µORB* topic implementation

The code generator (`genmsg_cpp`) transforms each topic file content in a structure that the compiler C/C++ can recognize. Such transformation follows a number of implementation rules set to enhance and manage the communication between modules/tasks. Before going into detail, let us examine the content of the topic file `follow_target` depicted in Figure 2.5 and its corresponding C/C++ header shown in Figure 2.6.

```
uint64 timestamp        # time since system start (microseconds)
float64 lat             # target position (deg * 1e7)
float64 lon             # target position (deg * 1e7)
float32 alt             # target position
float32 vy              # target vel in y
float32 vx              # target vel in x
float32 vz              # target vel in z
uint8 est cap           # target reporting capabilities
```

Figure 2.5: *The `follow_target.msg` file content*

```
1   struct follow_target_s
2   {
3       uint64_t timestamp;
4       double lat;
5       double lon;
6       float alt;
7       float vx;
8       float vy;
9       float vz;
10      uint8_t esc_cap;
11      uint8_t _padding[7];//required for logger
12  };
13  ORB_DECLARE(follow_target);//Registers the ORB struct.
14
```

Figure 2.6: *Exploring the `follow_target.h` header content*

The statement 12 of the code shown in Figure 2.6 is used for registering a topic metadata structure . In the present case this metadata is for the topic named `follow_target`. The structure of an `orb_metadata` [3] is shown in Figure 2.7. Each `orb_metadata` structure encapsulates another structure for storing the topic data values like the one depicted in Figure 2.6 namely `follow_target_s`. Each nested data structure must comprise the **`uint64_t timestamp`** field used for logging during the data publication. Additionally, the `_padding0[x]` field member is included to store the structure padding addresses [4].

[3] `orb_metadata`= object request broker metadata

[4] In order to align the data in memory, one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called structure padding.

2.2.2.3 The topic implementation cost

Within this section we show how expensive in terms of memory used to implement a single `orb_metadata`. For that purpose we use the `follow_target` topic as an example while considering the main structure of the `orb_metadata` which is presented on the Figure 2.7.

```
1       struct orb_metadata
2       {
3           const char *o_name;
4           const uint16_t o_size;
5           const uint16_t o_size_no_padding;
6           const char *o_fields;
7       };
8
```

Figure 2.7: *The structure of the orb metadata*

where

- `o_name` is a unique name assigned to each `orb_metadata`. In the present case this name is `follow_target`. The memory space allotted to store this name is equal to 14bits.
- `o_size` is the size of `follow_target_s`. The size required to store this structure including the padding addresses while no leaving free addresses between members (using of the pragma [5] directive) is 384 bits.
- `o_size_no_padding` is the object size without padding at the end needed. In result, 328 additional bits are required.
- `o_fields` is the semicolon separated list of fields with their types. The array of all fields in a semicolon separated list. In our example this chain of characters contains the following elements:
  `uint64_t timestamp;double lat;double lat;float alt;float vy;float vx;float vz;uint8_t est_cap`. This array costs 760 bits of memory space.

The memory space consumed by the only `follow_target` metadata is equal to 1486 bits, which is too much.

[5] `#pragma directive` can be used for arranging memory for structure members very next to the end of other structure members.

2.2.2.4 Communication management between modules

The purpose of this section is to analyze the easiness/complexity of the implementation of such a mechanism and its ability to comfortably satisfy the data property constraints considered in this thesis with respect to the real-time version of the PX4 system.

2.2.2.4.1 Notions and formalization

As previously mentioned, the μORB communications are based on shared files in which new data samples are published and from which subscribers read the necessary input data. The reading of the data is performed over an active pooling on the file, i.e. when a subscriber module task is triggered, it tries to read from each of its input files and, if there is no new data published since its previous completion, this task pins down until new data is produced. One of the consequences of this operating strategy is that it may lead to chain blocking, which is not desirable for real-time systems where the tasks are scheduled in a timely and predictable manner.

As for file management, within the μORB API, each topic owns a unique file used to store the data samples related to the corresponding metadata. Consequently, the system contains as many files as there are topics. These files are contained in a file map that we refer to as

$$\mathcal{F} = \{f_i, f_{i+1}, \cdots, f_{mf}\}$$

where f_i is the file related to the metadata of index i and mf is the maximum number of files that can be contained in $\mathcal{F}$. As a matter of fact, maximum number of files (topics) supported by the PX4 system is 365. Each f_i has the capacity of storing a number `qSize` of nodes at a time instant such that

$$f_i = \{node_{i,0}, \cdots, node_{i,qSize-1}\}$$

where $node_{i,j|j\in[0,qSize-1]}$ is the node located in the j^{th} part of f_i. A node consists in a set of a data sample and its handler. That is, each produced data sample is associated with its handler to form a node which is registered and published.

Accordingly, let $data_i^p$ be the p^{th} produced data sample related to the metadata of index i,

$\forall p \in \mathbb{N}^+$. Assuming that the handler of this data is `dHandler`, the corresponding is denoted by $node_i^p$ such that

$$node_i^p = \langle data_i^p, dHandler \rangle$$

For a good understanding of the meaning of $data_i^p$, we refer to the example presented in Figure 2.6 where the `follow_target` topic is considered. The corresponding metadata has a nested data structure named `follow_target_s` comprised of the following members: `timestamp`, `lat`, `lon`, `alt`, `vy`, `vx`, `vz`, `esc_cap` and `_padding0`. A data sample (i.e $data_i^p$) is made up of the values assigned to each of these variables except the one of `_padding0`.

2.2.2.4.2 The μORB data management life-cycle

The Figure 2.8 depicts the data state transition diagram for the μORB communication mechanism. The μORB data state transition diagram is composed of 3 main steps, namely *Node registering/unregistering*, *communication broking* and *data unpacking and reading*.

At the execution completion or during the context switch the module publishing the data samples related to the metadata of index i writes a new data sample, that we denote by $data_i^p$ where p stands for the p^{th} produced data, $\forall p \in \mathbb{N}^+$. The data handler generator associate to this data a handler `dHandler` that will be referencing this data throughout its existence. The combination of this data sample and its handler form a `node` which is copied into the corresponding waiting queue denoted by $pQueue_i$. In order to avoid redundant publications of a same node, by default, each node inserted into this queue has the publication status set `false`. A published node has the $pStatus$ status set to *true* and should no more be published. This is represented by $\langle node_{i,p}, pStatus \rangle$. At this same stage, all the nodes that were containing the data used by the latter during its execution are destroyed/unregistered.

The communication broker server periodically checks for new nodes and publishes them and, when a node is successfully published, all the subscribers are informed of the availability of this new data. From this moment, subscribers that are waiting for the new data to be published can start their executions (`Execution start stage`). Precisely, publishing a node consists in copying the node from the publication queue to the file associated with the corresponding metadata considering that each metadata owns a separate file. So for a

metadata of index i the file meant to store its nodes is presented as $sQueue_i$.

At the `execution start stage`, each module subscribes to all the topics it is supposed to subscribe to. So regarding the metadata of index i, the released module retrieves the node $node_i^s$ from the file $f_i \in Queue_i$. During the module execution time, if no new data samples are published (with respect to a topic/metadata), the corresponding file keeps the $uStatus_i$ [6] set to `false`, by default.

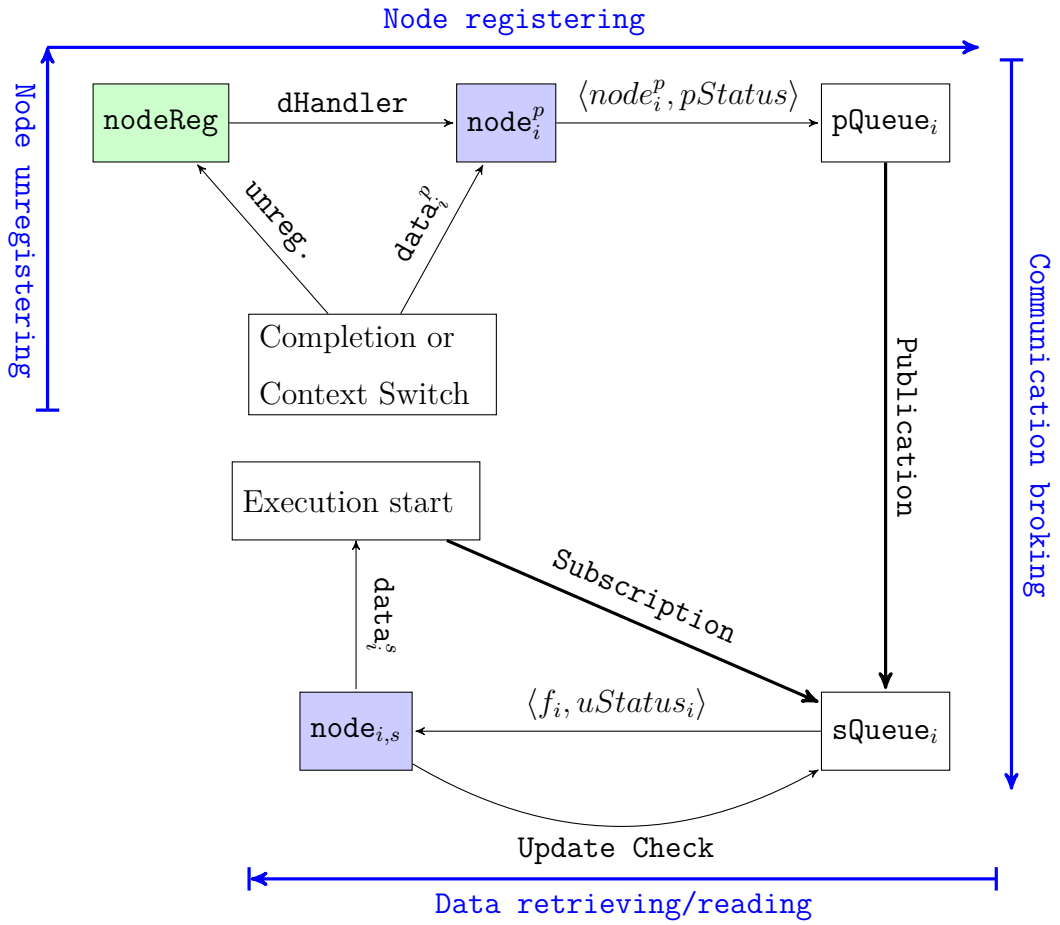

Figure 2.8: *The µORB data state transition diagram*

As for publications, this mechanism is used to monitor whether there have been any new publications since the beginning of the reading process. Therefore, with respect to the execution of a given module, at the end of reading, the module checks if there are no new publications regarding this metadata. With respect to this metadata and this module, if $uStatus_i$ is still `false` it shows that no new data have been published since the beginning of

[6] $uStatus_i$ is the update status of the file of index i

the previous reading and the module can continue processing the module codes. Conversely, if $uStatus_i$ status has changed to `true`, it shows that there have been new publications and the module must start reading again until the end of the reading happens before any new publication.

On the other hand, from the notation $node_i^s$, the value of s is the number of the node recently inserted into f_i. In fact, since there may be numerous nodes queued into f_i, s is the number of the last published node with $s \leq p$. At a time instant t, the equality $s = p$ shows that, since the reading of the s^{th} data sample (with respect to a given subscribing module), no other data sample is been produced after the registration of the node encapsulating the $data_{i,p}$ data sample. Similarly, if $s < p$, it means that there $p - s$ data samples registered into $pQueue_i$ (and probably having been published $sQueue_i$ and waiting to be consumed) since the beginning of the s^{th} data sample reading.

2.2.2.4.3 Data freshness and consistency maintenance

The data freshness property is easily guaranteed based on the data handler number. Indeed, each time a subscriber is triggered, it retrieves the node with the highest publication number (s) from each input queue ($sQueue_i$ for instance). Regarding the data consistency property, the μORB mechanism being managed `asynchronously`, this property ensured by implementing a `lock-free` mechanism [52]. No doubt about the feasibility of this mechanism in terms of maintaining this property and the μORB mechanism, in parallel to this `lock-free` mechanism, it uses a double buffering; separate buffer between a publisher and a subscriber [52].

However, as we are moving from a non-real time version to a real-time one, below we try to evaluate the efficiency of such an implementation, for a system subjected to strict real time constraints. In the real-time literature the *lock-free mechanisms* refers to the mechanisms that manage the shared resource in a way that each reader accesses the communication data without blocking and, at the end of the reading operation, it performs a check to see if no concurrent operations by the writer have happened in order to guarantees the data consistency without blocking on the shared resource. In the cases that these operations have taken place, the reader repeats the operation with the possibility of upper bounding these

retries [49, 51]. Although these tries can be bounded, it can only be possible if the system is predictable, which is not the case for the non-real time version of the PX4 system.

2.3 The thesis context

This thesis is carried out within the CEOS project intended to generate a real-time version of the PX4 autopilot system, named PX4-RT for *PX4 Real-time system*. PX4 is an autopilot developed by world-class developers from industry and academia, and supported by an active world wide community, it powers all kinds of vehicles from racing and cargo drones through to ground vehicles and submersibles. The PX4 structure consists of two main layers: the **flight stack** and the **middleware**. The flight stack is an estimation and flight control system while the middleware is a general robotics layer that can support any type of autonomous robot, providing internal/external communications and hardware integration. The transformation from PX4 to PX4-RT is carried out within the INRIA KOPERNIC team.

The PX4 system consists of a set of modules communicating by transmitting/receiving messages on the basis of publish/subscribe communication semantic. Speaking of real-time task scheduling, a module can be considered as a task. Each module is considered as a task in the NuttX operating system and is assigned a certain priority. The modules are scheduled according to a fixed priority scheduling algorithm. At each call of the scheduler, the module with the highest priority is executed. A running module can be preempted if there is a higher-priority module activated and returned to the ready queue.

Modules are scheduled based on the `first in, first out` policy. Moreover, the execution of the modules is subject to precedence constraints while being triggered according to a mixed mechanism provided that, by default, the modules are triggered periodically and, at release time, it is required to check if there are new data produced between the previous and current triggering time with respect to each of the topics it consumes (within PX4, almost each module produce and consume several topics). That is, if there is at least one topic for which no new output has been produced, the module waits (in an active polling) until there is a new data. During this waiting period, all modules waiting for data are blocked for the same reason (they can continue only when the module in progress will have produced new

data). Due to the lack of assurance or predictability regarding the data propagation, this operating mode constitutes a last option. The communication is done into three steps: **(i)** each module writes the data into a given buffer (at the execution completion or during the context switch) and, periodically, **(ii)** the written data are published (written into an other buffer from where the subscribers will read it) by a communication broker server, finally, **(iii)**, once the data are produced an advertising signal is sent to all the modules waiting form this data. Advertising topic consists in sending signal to all the subscriber that may be waiting for the venue of these data. From the above, it can be concluded that the PX4 system is designed on the basis of informatics and automatism theory, and therefore it cannot be predicted as required for a real time system.

Regarding the internal communications, PX4 uses the the so-called μORB (micro Object Request Brocker) communication mechanism, while the external communications are handled by the MAVLink or RTPS communication mechanisms. In this thesis the concern is on the internal communications. The μORB is an asynchronous messaging API following following the *publish/subscribe* paradigm.

2.3.1 The PX4-RT system

The original PX4 autopilot program is a set of modules. Each module is a thread managed by the NuttX operating system using a priority. The thread repeats itself indefinitely and activates the wait ("poll") on a data provided by another module. The modules that manage the sensors operate periodically and therefore periodically produce data for the other modules. These modules also operate periodically by inheriting periods from the modules managing the sensors and synchronizing with them via the data accesses they exchange. Also, the modules synchronize with each other in the same way.

Within the KOPERNIC team, studies and experiments aimed at transforming the PX4 system into a real time system (PX4-RT) have been carried out in the following way: the modules have been transformed into a set of dependent real time tasks, with temporal characteristics (activation periods and WCET as well as the deadline) according to the theory of real time task scheduling. Due to the fact that NuttX does not provide a function to create periodic tasks, the semaphores and the HRT function of PX4 have been used to periodically

lift interruptions. For each task, activated at a given period using an HRT function, an interrupt routine was created in which a semaphore is used to lock/unlock the task periodically, switching from the "locked" state to the "unlocked" and "ready to run" states. Each task has a period equal to its deadline. A priority was also assigned to each task, inversely proportional to its period, according to a "Rate Monotonic" scheduling algorithm. Each data exchange between modules represents a dependency between tasks.

2.3.2 The thesis goals

From the management and scheduling point of view, the transformation of the PX4 system to the PX4-RT has been successfully achieved. However, the PX4-RT system tasks continue to communicate using the *uORB*, which was not originally designed to be used with a system subject to strict timing constraints. This may not guarantee the data properties required for correct data quality which can only be ensured if, and only if, the communication does not induce additional constraints that would be inconsistent with the scheduling policy by avoiding any blocking semaphores on shared resources.

In this thesis, the aim is to propose a communication mechanism suitable for the PX4-RT system scheduling while taking into account the major specificities of the uORB communication. In order to guarantee a good data quality, the proposed communication solution must check a number of constraints to be verified through the preservation of a certain number data properties required based on the system requirement.

2.3.3 Identifying data constraints covered in this thesis

Before identifying the data constraints to be verified in this thesis, below we present some specificities regarding the communication model to be taken into account.

2.3.3.1 Communication system specifications and assumptions

Specification 1 *Communications between tasks is done via communication variables (non-mutual exclusive shared variables) which may be composed of several member variables. For instance, considering the data structure of message* `vehicle_attitude_setpointp`*, it is*

made of the following variables:

timestamp, *timeOfIssue*, *roll_body*, *pitch_body*, *yaw_sp_move_rate*, *yaw_body*, *q_d*, *q_d_valid*, *thrust_body*, *fw_control_yaw*, *roll_reset_integral*, *pitch_reset_integral*, *yaw_reset_integral*, *apply_flaps*, *FLAPS_LAND*, *FLAPS_TAKEOFF*, *and* *FLAPS_OFF*.

Specification 2 *The communication system is based on the "one producer, many readers" principle where only one task can write (modify) data while multiple consumers can read from the same communication variable. However, it is also possible that some tasks can produce data related to the same communication variable. In order to control who produced what when, each of these tasks will write into a separate location from the others. Therefore, each task consuming the data related to this communication variable will have to read from each of the locations.*

Specification 3 *The access to the communication variable is asynchronous with no blocking. That is to say that a task in progress of reading or writing can be preempted by the jobs of higher priority tasks and resume when it returns into active mode. Each task writes or reads data based on its priority.*

Specification 4 *The communication between the producer and consumers of the data is isolated in the sense that the producer is not aware of who consumes the data it produces likewise the consumers are not aware of the producers of the data they use.*

Specification 5 *A task can produce data related to different messages. The data related to all these messages are not mixed; each of these messages has a separate communication variable. Then, each of the communication variables can be accessed by different consumers. As a result, through a given communication variable, a task may be in communication with a different number of consumers compared to those in communication through another communication variable.*

Specification 6 *A job of each task reads the input data at the activation and writes the output data at its execution completion. Thus, unlike the LET model [62], once a data is successfully written, it becomes available and any task activated immediately can consume it.*

Data access is explicit, meaning that data is read directly from the main memory. No shared variables local copies are required, contrary to the implicit communication model [7] proposed in AUTOSAR.

2.3.3.2 Data constraints and properties identification

In this section we describe the constraints to be satisfied to guarantee the quality of the data used in an embedded real-time system. Speaking of data quality, it is translated into a list of properties that the data must have and on which basis it is possible to determine the correctness of such data in accordance with the system requirements. Accordingly, in this thesis, we consider four data properties, namely, `data consistency`, `data local coherence`, `data freshness` and `data global matching`.

Property 1 (`Data consistency`) *In accordance with the Specification 1, the data consistency property of a data related to a communication variable is ensured if, and only if, each job of the consumer tasks uses exclusively (during its execution step) the values resulting from the same execution step of the producer.*

This property covers the `consistency in value` and `consistency with other variables` as presented in Figures 1.8 and 1.9, respectively. In other words, for a communication variable comprised of several variables receiving each a data at the execution completion of the producer task, the consumer should only consume the values (per variable) produced at the execution completion of the job of the data producing task. Otherwise, this would lead to an overlapping situation where an instance resuming after the production of new data is likely to use part of the old data together with part of the new data produced during the preemption period.

Property 2 (`Data freshness`) *For a task consuming data related to one or several communication variables, the data freshness property is maintained if this task always consumes only recently written data by the producer for each communication variable.*

Property 3 (`Data local coherence`) *According to Specification 2, if the data related to a communication variable are produced by different tasks, local data local coherence property is*

ensured, if and only if, for each of the related communication variables, the consumer task reads data verifying freshness property (Property 2).

Property 4 (`Data global matching`) *This property derives from the end-to-end properties forming a spindle. Thus, the global data matching property is ensured if and only if the* ***`spindle sink`*** *task always reads data samples resulting from the same execution step of the* ***`spindle source`*** *task.*

Guaranteeing the data matching property may be an indispensable mandatory requirement for smart systems (drone autopilot, autonomous vehicles,$\cdots$) which make driving decisions based on the sensed information. For a correct functioning of the system in accordance with its requirements, several of these properties aforementioned can be required all together or separately. Consequently, to guarantee a good quality of data both locally (by requiring task-to-stain properties) and globally (by requiring end-to-end properties), it is necessary to verify a couple of constraints, namely, local consistency data constraint and global consistency data constraint, defined below:

Constraint 1 (`Local consistency constraint`) *Considering two tasks τ_i and τ_j communicating through a shared resource r such that $\tau_i \xrightarrow{r} \tau_j$ a local consistency constraint denoted* ***LocalCC***$\langle \tau_i, r, \tau_j \rangle$ *is verified if, all the data samples written by τ_i into r, the jobs of τ_j always read data samples that are consistent, fresh and locally coherent.*

Precisely, a data sample is said to be `consistent` if it has the property 1, `fresh` if it has the property 2 and `locally coherent` if it has the property 3.

Constraint 2 (`Global consistency constraint`) *We consider two tasks τ_{src} and τ_{sink} and q functional chains propagating the data produced by τ_{src} until τ_{sink} forming a spindle $\mathcal{S}(\tau_{src}, \tau_{sink})$ where τ_{src} and τ_{sink} are the spindle source task and the spindle sink task, respectively. Let r_c be a resource shared between the second-to-last task of a chain of index c and τ_{sink} provided $1 \leq c \leq q$. The global consistency constraint, denoted by* ***GlobalCC***$\langle \tau_{sink}, \{r_c\}_1^q \rangle$, *is verified if the jobs of τ_{sink} always read from all the q resources the data samples having the Property 4 such that for each of the data read from r_c is Property 1-guaranteed.*

Regarding the global matching constraint, it is essential to preserve the consistency of each data since we have no arbitration mechanism for the access to the communication variables. Therefore, the sizes of the buffers at the end of each spindle chain have to take into account not only the global matching property (Property 4) but also the consistency (Property 1) of the data exchanged between each second-to-last task of each chain and the spindle sink task.

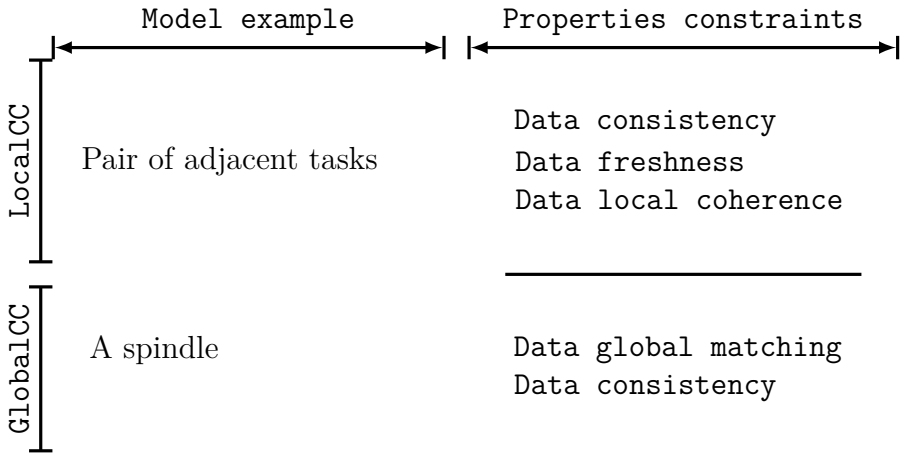

Figure 2.9: *Data constraints and properties overview*

Part II

CONTRIBUTIONS AND RESULTS

Chapter

3 Modeling and formalization

Chapter content

3.1 The tasks system model 60

3.2 The communication model 62

3.2.1 A basic organization of the buffer 62

3.2.2 Message characterization 65

3.2.3 Data sample characterization 67

3.3 The communication graph 69

3.3.1 Shared variables management 71

3.3.2 Inter-task communication relations 72

We present within this chapter our first contributions of this thesis. We begin by presenting our model of tasks systems and the inter-tasks communication model.

3.1 The tasks system model

We consider a periodic time-triggered system consisting of a set $\mathcal{T}$ of n tasks executing upon a uni-processor or partitioned multiprocessor platform. The tasks are independent and scheduled preemptively based on a fixed-priority scheduling policy such as rate monotonic [2] or deadline monotonic [10]. As shown previously, the system predictability implies both the scheduling and the communication predictability. To that end, regarding the system scheduling, additionally to meeting the deadline requirements, Ndoye and Al. in [108, 109],

modify the Liu and Layland's model [2] by adding a `scheduling state` parameter (further denoted by θ) indicating if the job to be executed belongs to the task that is in a sleeping state or if this job exits the preemption state in order to calculate the cost of the preemptions. Precisely, the task execution state value is 1 if it is a new job of τ_i and 0 if the job of τ_i resumes from a preemption. However, it should be noted that this enhancement gives no information about the state of writing or reading data. In order to enhance the inter-tasks communication predictability, we consider a task `communication state` parameter that we denote by ϖ. Thus, each task τ_i is now described by the tuple $(c_i, C_i, T_i, \theta_i, \varpi_i)$, where c_i and C_i are the best- and the worst-case execution times, T_i the period, θ_i the scheduling state and ϖ_i the task communication state. We assume that all the tasks are released simultaneously, have implicit deadlines ($T_i = D_i$), and no offset is considered ($O_i = 0, \forall i$). The parameter ϖ_i may have the following values depending on the task communication state: (i) $\varpi_i = -1$ corresponds to the state when the job of τ_i is busy with reading the required inputs or no job of τ_i is executing at all (τ_i is sleeping), (ii) $\varpi_i = 0$ when the job of τ_i has finished reading the input data and is busy with processing the task program, including when the job is preempted by the jobs of the higher priority tasks and (iii) $\varpi_i = 1$ when the job currently executing is busy with writing the output data. In order to control the writing instants, at the end of the task program, we add the `endProcc` instruction line. Once this line is executed, it triggers the writing of the output data. With the help of this parameter no job is able to write during the context switch or from any other point within the job execution window.

Basically, each job is supposed to start its execution (at its activation) by a reading from all the input resources and completes its execution by a successful writing into the output resources. The job execution completion corresponds to the time instant when an output data is correctly inserted into the corresponding resource. Only at the job execution completion instant the value of ϖ_i changes from `1` to `-1`. The exact execution duration of the p^{th} job of τ_i, denoted by C_i^p is such that $c_i \leq C_i^p \leq C_i$. The execution of a job is considered as the sum the CPU time units spent on the reading of input data, the processing time of the task program and the time spent on writing output data. In Figure 3.1 the reading time interval (`r`) is represented by the blue box, the processing time (`p`) by the gray box while the writing time (`w`) is represented by the yellow box. The upper plot describes a situation where

the task job executes for its best-case execution time (c_i) while the down plot concerns the worst-case execution time (C_i).

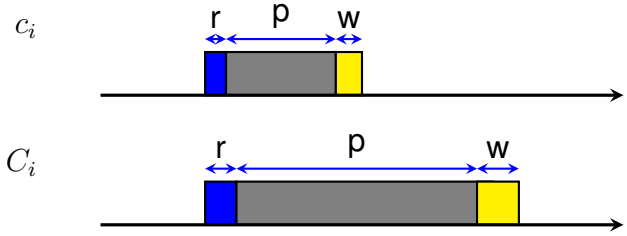

Figure 3.1: *The task execution time is the sum* `r+p+w`

Without any loss of generality, we consider that the tasks are ordered from the highest priority to the lowest priority. Hence, if $i < j$, then τ_i has a higher priority than τ_j.

3.2 The communication model

The tasks communicate through the bounded FIFO circular buffer known to be a `FIFO data structure` that considers memory managed cyclically and it may offer several advantages. For instance, contrarily to the sequential buffer, no shifting of the data from slot to slot is required; each data sample remains in the initial slot until it is overwritten. Thus, depending on the size of the buffer, the largest time a data can remain into the buffer before being overwritten is referred to as the `data lifetime bound` [54, 58]. Our choice of the circular buffer is motivated by the fact that it guarantees a deterministic data management.

3.2.1 A basic organization of the buffer

We understand by a `buffer`, a reserved memory space storing communication data related to a given message. The communication between tasks is ensured via a communication resource denoted by ∇. In this thesis a communication resource is a circular buffer. We associate a communication resource ∇^m to each message m and we characterize it by following tuple $(\mathcal{D}^m, size, head, tail)$, where:

- $\mathcal{D}^m$ is an array containing the data carried by the message m.
- $size = |\mathcal{D}^m|$ is the cardinality of $\mathcal{D}^m$, for any message m. So, $size$ is the maximum number of different data copies that can be hold into $\mathcal{D}^m$ at the same time. Each copy

of the data, referred to as `data sample`, is denoted by ∂^m.

Each data sample is constituted by the values assigned to all variables forming the structure of the corresponding message. Each ∂^m occupies a single slot (cell) in $\mathcal{D}^m$. Slots are numbered from $0 \rightarrow size - 1$, where $size$ is fixed for each ∇^m and assigned at the system run. We do not consider any dynamic memory allocation.

- *head* is an integer indicating, to the running job of a given producer task of a message m, into which slot (in $\mathcal{D}^m$) to write the output result at the end of the execution.
- a *tail* value indicates, for the tasks reading from $\mathcal{D}^m$, from which slot of $\mathcal{D}^m$ to read.

The values of `tail` and `head` indicate the buffer locations respectively for reading and writing operations and loop back to 0 after their values reach the size of the buffer [58, 110, 111].

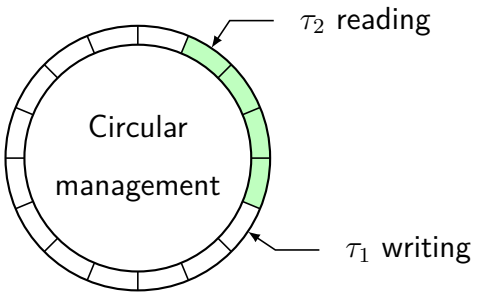

Figure 3.2: *An example of a circular buffer shared between two tasks*

The access to the communication resources follows the `single writer many readers principle`, where only one task can write into a given communication resource while several tasks may concurrently read from it. Each reader task owns its separate "tail" value. Hence, there are no dependencies with other reading tasks. In fact, this principle allows two or more tasks to perform the reading action from different slots at a given time instant, thus, a purely asynchronous communication is ensured.

The circular buffer offers the possibility to each of the `q` consumers to read from different slots at a same time instant. Hence, `tail` is a array of q integer values. The value of `tail` pointer indicates to each of the q consumers which slot $s_l \in \{s_0, \cdots, s_{size-1}\}$ to access for the reading action. Thus, at the activation time, a job of each task $\tau_c \in \{\tau_1, \cdots, \tau_q\}$ reads from ∇ the data sample of the buffer slot pointed by the current value of its `tail`. After a successful read, the current value of `tail` is incremented such that $tail_{c,current} \leftarrow (1 + tail_{c,previous})$

mod *size*. Likewise, the value of `head` is incremented at each execution completion of a job of τ_p in the following way: $head_{current} \leftarrow (1 + head_{previous}) \mod size$. This new value of `head` points to the slot where the next job of τ_p writes at its completion. Finally, in the essence, the circular buffer is organized in a way that the jobs of each $\tau_c \in \{\tau_1, \cdots, \tau_q\}$ should consume all the produced data from slot to slot.

For instance, let us consider the model presented in Figure 3.3 where the task *gps* produces data samples to be used by the tasks *cmdr* and *nav*.

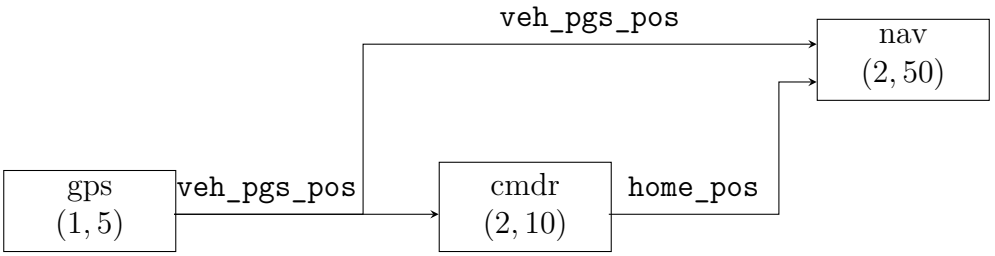

Figure 3.3: *Tasks used to check the age of the data within the circular buffer*

In this example, we assume that the data samples related to the messages `veh_gps_pos` and `home_pos` are written to communication resources named ∇_1 and ∇_2, respectively. Furthermore, each task is described by its worst-case execution time and a period. Tasks are scheduled according to the rate monotonic policy [2]. Assuming that ∇_1 and ∇_2 are sufficiently large, the sixth job of *cmdr* consumes the sixth data sample produced by *gps* while the second job of *nav* consumes the second data sample. However, by the time the jobs in *cmdr* and *nav* are activated, 5 and 9 data samples are added into communication resource.

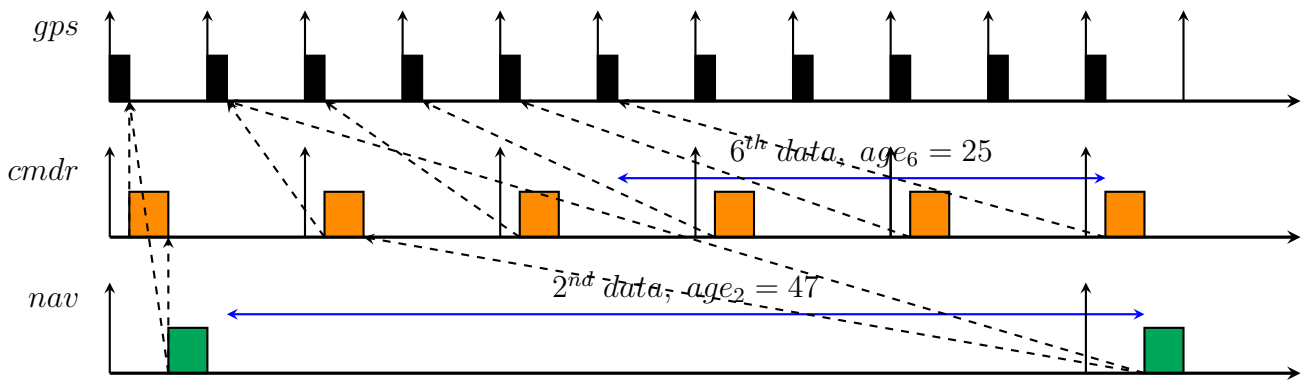

Figure 3.4: *Example displaying the age of the data within the circular buffer*

Consequently, this type of data management results in larger data ages (25 and 47, respectively) and may not be suitable for systems requiring newly generated data. Thus, `will such data still be useful for the functional correctness of the system?` Later in this thesis (Section 4.2), we propose a mechanism adapting the circular buffer to the constraints imposed in embedded real-time systems.

3.2.2 Message characterization

Compared to the original PX4, the PX4-RT drone autopilot system keeps the same μORB communication mechanism. The latter considers a message or topic as the communication unit. As indicated previously, the μORB mechanism supporting the PX4 system [52] does not fulfill the requirements of a real-time system.

Therefore, although PX4-RT is a real-time system which guarantees the schedulability of the system of tasks, unfortunately, it doesn't ensure a quality of the data shared by these tasks. Basically, this conclusions motivates the contribution presented in this thesis: the proposition of a communication mechanism (API) guaranteeing the quality of the data within the PX4-RT and similar systems.

The novel communication mechanism keeps the `message` as the communication unit and it proposes data management policy different from `publish/subscribe` paradigm. We start by formalizing all the communication particularities regarding the organization of the message as the communication unit.

The set of all messages shared by the tasks of $\mathcal{T}$ is denoted by $\mathcal{M}$ and the set of the input messages and, respectively, the output messages for each task τ_i, is denoted by $\mathcal{M}_i$, provided that $\mathcal{M}_i \subset \mathcal{M}$ and $\mathcal{M}_i = \mathcal{M}_i^I \cup \mathcal{M}_i^O$. We denote by $\mathcal{M}_i^I$ and $\mathcal{M}_i^O$ the input and output messages for the task τ_i, respectively.

Definition 1 (`Input message,` $\mathcal{M}_i^I$) *:* $\forall(\tau_i \in \mathcal{T} \wedge \mathcal{M}_i \subset \mathcal{M})$, $\mathcal{M}_i^I$ *is given by* $\mathcal{M}_i^I = \{m_i^{I,1}, \cdots, m_i^{I,nI}\}$ *where* nI *is the total number of input messages for* τ_i*.*

Analogically we define the τ_i output messages as follows:

Definition 2 (`Output message,` $\mathcal{M}_i^O$) *:* $\forall(\tau_i \in \mathcal{T} \wedge \mathcal{M}_i \subset \mathcal{M})$, $\mathcal{M}_i^O$ *is given by* $\mathcal{M}_i^O = \{m_i^{O,1}, \cdots, m_i^{O,nO}\}$ *where* nO *is the total number of output messages for* τ_i*.*

The Figure 3.5 presents the input and output messages regarding the task `position_ctrl` where, on the left side, we have the list of input messages and, on the right side, the output messages. Here `position_ctrl` has 11 different input messages (`nI`=11) and 6 different output messages (`nO`=6).

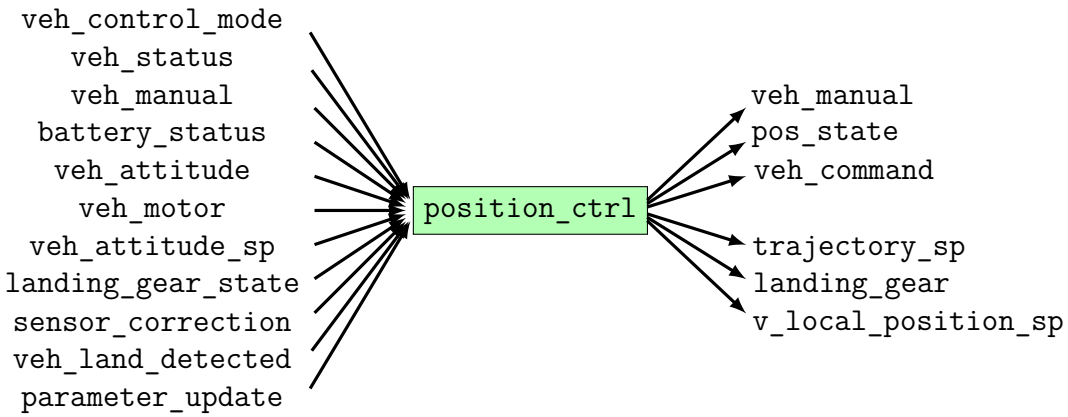

Figure 3.5: *Example of input and output messages for the task* ***`position_ctrl`***

We implement each message as a unique buffer managed following the `single writer many readers principle` as previously mentioned. However, within PX4-RT, we have cases where the same message is produced by different tasks. An example of such a case is presented in Figure 3.6 where the message `vehicle_attitude_setpointp` is produced by three different tasks, namely, `mc_position_ctrl`, `gnd_position_ctrl` and `navigator`.

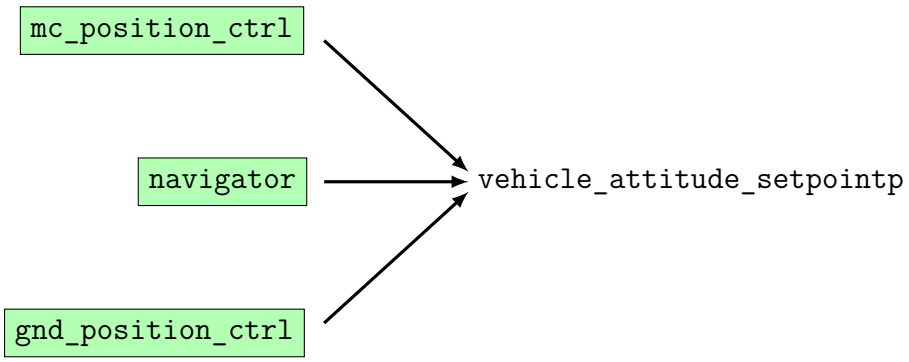

Figure 3.6: *An example of a multi-source message*

On this basis, all the messages are classified into `single-source` and `multi-source` messages.

Definition 3 (`Uni-source message`) *A message is said* ***`uni-source`*** *if it is produced by a single task.*

Definition 4 (`Multi-source message`) *A message is said `multi-source` if it is produced by more than one task.*

One of the unaddressed cases in the μORB mechanism regards the `multi-source message` data management. Precisely, as we can read from the μORB documentation, **"any number of advertisers may publish to a topic, publications are atomic but co-ordination between publishers is not provided by the ORB"** [1]. Subsequently, guaranteeing the properties like `data consistency` is quite impossible in such working conditions.

3.2.3 Data sample characterization

Each message is a data structure comprising several variables as depicted in Figure 3.7. The data carried by a a message m are stored into the corresponding $\mathcal{D}^m$ which has a fixed size $|\mathcal{D}^m|$. Each data sample $\partial_i^m \in \mathcal{D}^m$, provided $1 \leq i \leq |\mathcal{D}^m|$, is constituted by the values assigned to each of the variables forming the structure of the corresponding message.

```
uint64 timestamp                # time since system start (microseconds)
float32 roll_body               # body angle in NED frame (can be NaN for FW)
float32 pitch_body              # body angle in NED frame (can be NaN for FW)
float32 yaw_body                # body angle in NED frame (can be NaN for FW)
float32 yaw_sp_move_rate        # rad/s (commanded by user)
float32[4] q_d                  # Desired quaternion for quaternion control
bool q_d_valid                  # Set to true if quaternion vector is valid
float32[3] thrust_body          # Normalized thrust command in body NED frame [-1,1]
bool roll_reset_integral        # Reset roll integral part (navigation logic change)
bool pitch_reset_integral       # Reset pitch integral part (navigation logic change)
bool yaw_reset_integral         # Reset yaw integral part (navigation logic change)
bool fw_control_yaw             # control heading with rudder (used for auto takeoff on runway)
uint8 apply_flaps               # flap config specifier
uint8 FLAPS_OFF = 0             # no flaps
uint8 FLAPS_LAND = 1            # landing config flaps
uint8 FLAPS_TAKEOFF = 2         # take-off config flaps
```

Figure 3.7: *The `vehicle_attitude_setpointp` message*

For instance, considering the example presented in Figure 3.7, each data sample carried by the message VAS (`vehicle_attitude_setpointp`), ∂^{VAS} is constituted by the values assigned to all the 16 variables composing the structure of the message VAS.

[1] This statement in the double quote is taken from the μORB documentation

Notice 6 *In this thesis, to accurately track the propagation of different data samples we consider that each data sample is double time-stamped. To that end, we use two parameters named* `timestamp` *and* `timeOfIssue`*. From the fact that each data sample in propagation has a nearby or far away origin, the* `timestamp` *is the intrinsic date of birth of a data sample. It assigned by the sensor task. This date is not modified throughout the propagation until the data sample is overwritten. Therefore, each job executing using a time-stamped data, indicates the output date of a data sample. we call this date the* `timeOfIssue`*. For the sensor task,* `timestamp=timeOfIssue`*.*

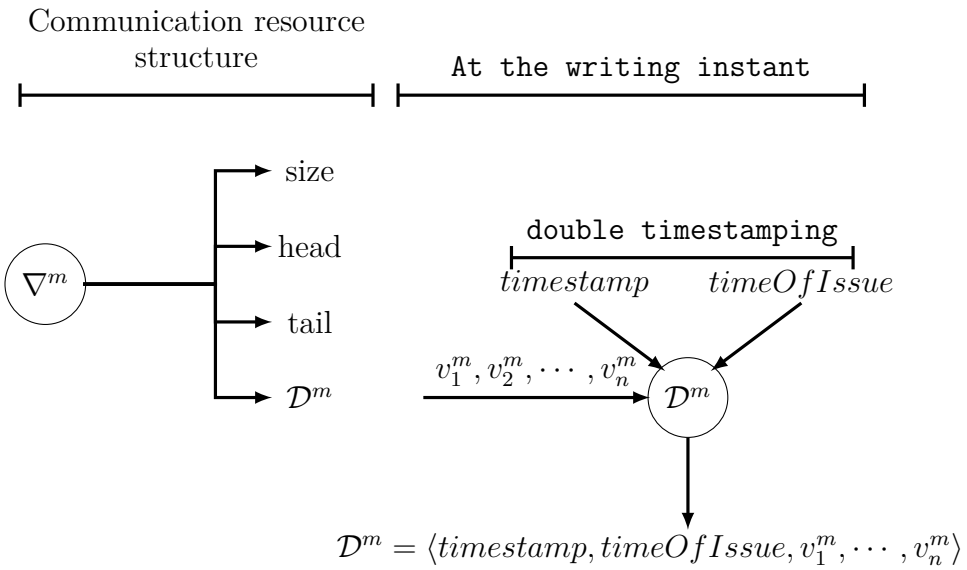

Figure 3.8: *Example of a data double stamping process*

3.3 The communication graph

One of the problems complicating the management of the data within the PX4-RT system resides in the fact that a single task can produce numerous different messages where each of the messages may in turn be consumed by several tasks and so on and so forth. This results in a complex network of connections between tasks. For a real-time system, such communicating tasks are mostly represented in the form of a directed acyclic graph (DAG) [112, 113]. Therefore, a DAG can be used to represent both the `scheduling order` and the `communication order` for a set of tasks. Regarding the scheduling order, for a pair of tasks (τ_i, τ_j) such that $\tau_i \rightarrow \tau_j$, the DAG indicates the precedence constraints between τ_i and τ_j; meaning that the execution of τ_j can start only if τ_i has completed. Also, when the DAG is used to show the communication order, $\tau_i \rightarrow \tau_j$ indicates that τ_j uses the data produced by τ_i.

Since there may be cycles between two tasks, the PX4-RT system cannot be represented by a classical DAG. Subsequently, in order to explicitly consider all the possible communications between tasks, we propose to model the PX4-RT by a bipartite graph [114] consisting of two disjoints sets, namely $\mathcal{T}$ and $\mathcal{M}$ such that every edge connects a vertex (task) in $\mathcal{T}$ to one in $\mathcal{M}$ and $\mathcal{E}$ is the set of linking edges. By doing so, we discard any direct communication between tasks while keeping the graph acyclic. Henceforth, the tasks in $\mathcal{T}$ can only communicate through the messages in $\mathcal{M}$ in a way that the producer task is not aware of the consumers of the messages it produces and vice versa. Accordingly, the access to the message buffers is `fully asynchronous non-blocking` in the sense that each task reads and writes on its own pace regardless of the execution state of the other tasks.

Finally, the resulting graph is referred to as the `communication graph` and an example of such a graph is presented in Figure 3.9. Herein, tasks are represented by ellipse and messages by rectangles.

Definition 5 (`The communication graph`) *The communication graph is a bipartite graph $\mathcal{G} = (\mathcal{T}, \mathcal{M}, \mathcal{E})$ such that $\mathcal{T}$ is the set of tasks, $\mathcal{M}$ is the set of messages and $\mathcal{E}$ is the set of linking edges between the tasks in $\mathcal{T}$ and the messages in $\mathcal{M}$.*

In the example depicted in Figure 3.9 it is shown that τ_1 communicates with τ_2, τ_3, τ_4 and

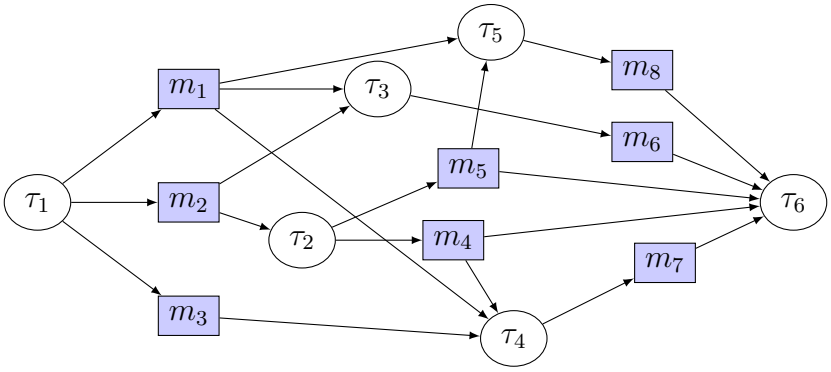

Figure 3.9: *An example of the communication graph*

τ_5 by means of the message buffers m_1, m_2 and m_3. Due to the `one writer, many readers` principle and to the `data lifetime method` calculating the sizes of different buffers, the fact that these four readers may have different worst-case response times leads to the conclusion that the corresponding buffers may also be of different sizes. Without any other arbitration mechanism, a sufficient size, also referred to as `optimal` (Def. 13), is the one guaranteeing that all the data, which may be produced during the execution of a job of each of the consumers, must not overwrite the data used at the execution start. In order to cover all scenarios by formally being able to calculate the sizes of the buffers used by the producer based on timing parameters of the consumers reading the input data from a given buffer, we introduce the concept of `domain message` as follows:

Definition 6 (`Message domain`) *The message domain denoted by $\mathcal{E}^*$ is the set comprised of a producer task and the consumer tasks for a given message, where the asterisk represents any message.*

***Notation**: $\mathcal{E}^*\{producer|consumers\}$*

For instance, for the communication graph presented in Figure 3.9, we obtain the following message domains (Figure 3.10):

In the remainder of this thesis, we use the general notation; $\mathcal{E}^*$ without specifying the message name. To each message domain $\mathcal{E}^*$ corresponds a unique buffer that we denote by ∇^*.

$$\mathcal{G} = (\mathcal{T}, \mathcal{M}, \mathcal{E}) = \begin{cases} \mathcal{E}^{m_1} = \{\tau_1 | \tau_3, \tau_4, \tau_5\} \\ \mathcal{E}^{m_2} = \{\tau_1 | \tau_2, \tau_3\} \\ \mathcal{E}^{m_3} = \{\tau_1 | \tau_4\} \\ \mathcal{E}^{m_4} = \{\tau_2 | \tau_4, \tau_6\} \\ \mathcal{E}^{m_5} = \{\tau_2 | \tau_5, \tau_6\} \\ \mathcal{E}^{m_6} = \{\tau_3 | \tau_6\} \\ \mathcal{E}^{m_7} = \{\tau_4 | \tau_6\} \\ \mathcal{E}^{m_8} = \{\tau_5 | \tau_6\} \end{cases}$$

Figure 3.10: *Example of the message domains*

3.3.1 Shared variables management

To correctly ensure the data integrity asynchronously without any arbitration mechanism for accessing data, only the structural variables (`tail`,`head` and `size`) are openly shared, while, regarding the data variable `data variable` (∇), a pointer to the memory space, where it is located, is shared.

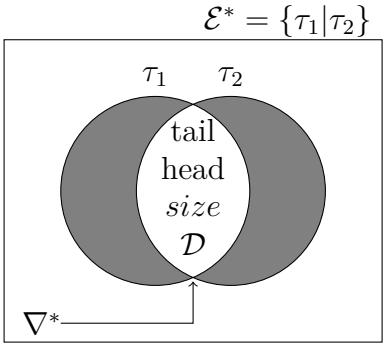

Figure 3.11: *An example displaying communication variables shared between adjacent tasks*

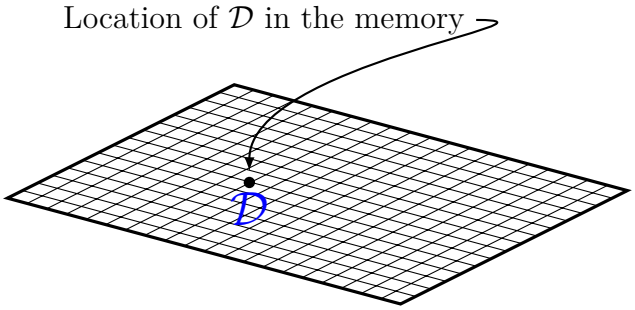

Figure 3.12: *Example showing the memory space allocated to the data variable $\mathcal{D}$ (the first bit of the first byte). $\mathcal{D}$ occupies contiguous addresses*

The grey circles in Figure 3.11 represent each task code program. The white part shows the common or shared elements. Within the same message domain, the producer uses the value of `head` to target the cell where the next output should be written. In other words, the recently produced value is located into the slot number $(currentValueOfHead(\nabla^*) - 1) \mod |\nabla^*|$. The value of `head` increments after a successful writing of the data and, at this instant, its current value is given by $(previousValueOfHead(\nabla^*) + 1) \mod |\nabla^*|$. On the

other hand, each consumer reads the data in the cell pointed by the current value of his own `tail`. The value of `tail` increments after a successful reading with respect to a given input buffer; when a task reads from different buffers, after a successful read from each of them, the corresponding value of `tail` is updated. So, at that instant, the current value of `tail` is given by $(previousValueOfTail(\nabla^*) + 1) \mod |\nabla^*|$. Therefore, when it happens that a producing task preempts a consuming one, or when the preemption is performed by another consumer, there is no blocking on the data variable, since the access routes do not cross.

3.3.2 Inter-task communication relations

On the basis of a message domain organization, the communicating tasks may be in `linear relation` or a `fork relation` as defined in the Definitions 7 and 8 , respectively.

Definition 7 (`Linear relation`) *We consider the message domain $\mathcal{E}^*$ such that $\mathcal{E}^* = \{\tau_i|\tau_{j_1}, \cdots, \tau_{j_k}\}$. A `linear relation` exists between τ_i and $\{\tau_{j_1}, \cdots, \tau_{j_k}\}$ if $k = 1$.*

Analogically,

Definition 8 (`Fork relation`) *We consider the domain $\mathcal{E}^*$ such that $\mathcal{E}^* = \{\tau_i|\tau_{j_1}, \cdots, \tau_{j_k}\}$. A `fork relation` exists between τ_i and $\{\tau_{j_1}, \cdots, \tau_{j_k}\}$ if $k > 1$.*

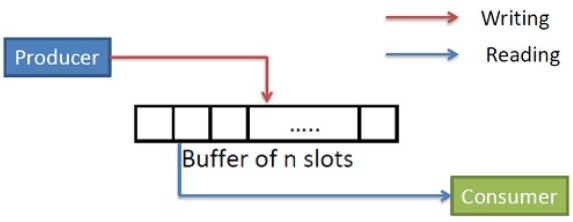

Figure 3.13: *A `linear relation`*

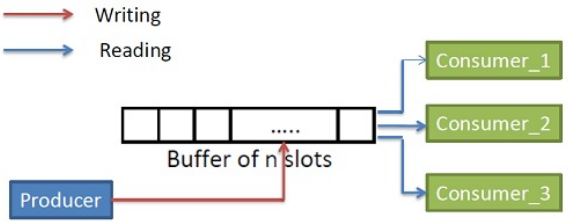

Figure 3.14: *A `fork relation`*

Not all the vertices in $\mathcal{T}$ have corresponding vertex in $\mathcal{M}$. Accordingly, we classify the tasks in `sensor, internal`, `sensor tasks` (Def. 9), `middle tasks` (Def. 11) and `actuator tasks` (Def. 10).

Definition 9 (`Sensor task`) *A task τ_i is a sensor task iff $\mathcal{M}_i^{In} = \varnothing$.*

Definition 10 (`Actuator task`) *A task τ_i is an actuator task iff $\mathcal{M}_i^{Out} = \varnothing$.*

Definition 11 (`Inner task:`) *A task τ_i is an inner task iff $\mathcal{M}_i^{In} \neq \varnothing \wedge \mathcal{M}_i^{Out} \neq \varnothing$.*

Considering the communication graph presented in Figure 3.9, τ_1 is a sensor task, τ_6 is an actuator task while τ_2, τ_3, τ_4 and τ_5 are middle tasks. Each pair of producer and consumer tasks belonging to a same message domain are said to be adjacent and defined as follows.

Definition 12 (`Adjacent tasks`)
Two tasks τ_i and τ_j are adjacent iff $\mathcal{M}_i^{Out} \bigcap \mathcal{M}_j^{In} \neq \varnothing$.

The set of messages produced by τ_i and consumed by τ_j is given by $\mathcal{M}_i^{Out} \bigcap \mathcal{M}_j^{In}$. We note that two tasks may be adjacent on different messages.

Chapter

4 Local consistency constraints

Chapter content

4.1 Computing the buffers optimal size 76

4.2 The sub-sampling rate mechanism 80

4.2.1 Introduction 81

4.2.2 Formalization and implementation 82

4.3 On the verification of local consistency constraints 88

4.3.1 Data freshness 88

4.3.2 Data local coherence 89

4.3.3 Data consistency 89

In this chapter we present our solutions regarding the data properties constraints formalized in the previous chapter. To this end, we recall the main assumptions considered herein: **(i)** the communication between the producer and the consumer tasks is isolated; that is, the producer task is not aware of the tasks consuming the data it produces and vice-versa, **(ii)** the communications between producers and consumer tasks is ensured through shared message buffers with fixed size, **(iii)** writing and reading the data into/from the shared buffers is asynchronous non-blocking; that is, each task accesses the shared buffer on its own pace based on its scheduling priority, **(iv)** the access to the shared buffers is based on the `single writer many readers principle` and **(v)** finally, each task reads all the required input data at the activation time and writes the output data at the execution completion.

Constraints related to data properties are taken into account starting with those required

at the local level (task level) and ending with those required at the global level (functional chain level). We show that verifying the global properties, also, maintains the task-level properties constraints. For each property level (task-/functional chain-), the verification of the required constraints follows the following steps: **(i)** computing the optimal size (Def. 13) for the inter-task shared buffer, **(ii)** proposing a suitable mechanism taking into account the assumptions considered in this thesis and, **(iii)**, lastly, proposing an algorithm capable of efficiently implementing this mechanism.

Definition 13 (Buffer optimal size) *We consider a communication resource* ∇^* *such that* $\exists \mathcal{E}^* = \{\tau_i | \tau_{i_1}, \cdots, \tau_{i_k}\}$ *where* k *is the number of consumer tasks. The optimal size of* ∇^**, denoted by* $|\nabla^*|$*, is the smallest size guaranteeing that any data sample read from* ∇^* *is never overwritten before the execution completion of the job(s) of any of the* $\tau_{i_1}, \cdots, \tau_{i_k}$ *tasks still processing this data.*

Computing such buffer size referrers to the *data lifetime bound method [54, 58]*. The latter relies on the computation of an upper bound of the number of times the writer can produce new values while a given data sample is being used by at least one reader. Given that a task can produce several different messages which can be consumed by different readers having different time periods, then the output buffers for a given task can have different sizes. Therefore, in order to efficiently calculate the required size for each of the buffers, we refer to the corresponding message domain indicating in a precise and clear way which are the tasks accessing this buffer. For instance, let us consider the system below where the task τ_1 produces two messages m_1 and m_2 shared with τ_2 and τ_3 as shown in Figure 4.1. Here the message domains are as follows: $\mathcal{E}^{m_1} = \{\tau_1 | \tau_2\}$ and $\mathcal{E}^{m_2} = \{\tau_1 | \tau_3\}$. The Figure 4.2 presents a situation regarding a multi-source message. A same message m is produced by two different tasks τ_1 and τ_2. As previously mentioned, in such a situation each producer writes into a separate array vector of the same bi-dimensional array buffer that we denoted, respectively, by $\nabla_{1,1}$ and $\nabla_{1,2}$. So, depending on the periods of τ_1 and τ_2, $|\nabla_{1,1}|$ and $|\nabla_{1,2}|$ may be different. The corresponding message domain becomes $\mathcal{E}^* = \{\tau_1, \tau_2 | \tau_3\}$.

Considering that the communication resources consist of composite data structures, the local coherence and freshness constraints require that only recently produced data should

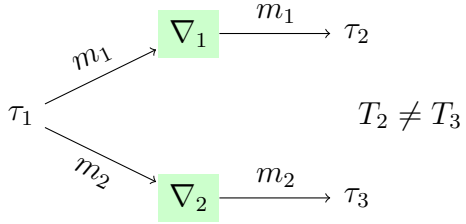

Figure 4.1: *Example of two buffers written by a same task but likely to have different sizes.*

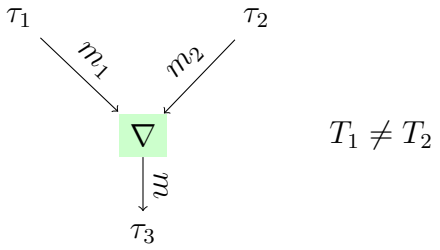

Figure 4.2: *Example of a bi-dimensional multi-source buffer where each of the vectors may have different sizes.*

be entirely consumed (only the data values resulting from the same execution step of the producer can be considered together) given that the communication buffer is managed cyclically and may have a storing capacity of more than one data sample, whether the latter is single- or multi-source one. Obviously, given the circularity of the buffer, it is impossible to implicitly know in which slot such a data is located (the one fulfilling the property constraint), as it can be in any of the buffer slots; and not at the front or back as in sequential buffers. Finally, for multi-source messages, it must be ensured that the reader task retrieves the recently produced data from each vector of the two-dimensional array.

In the first part of this section, we calculate the required sizes for each of the buffers independently of each of the data property constraints mentioned above; we refer only to the data lifetime bound method. Throughout the second part, we propose a mechanism that, while taking into account the calculated sizes, guarantees the maintenance of all the constraints considered at this level. This mechanism is called `sub-sampling rate mechanism`. An algorithm implementing this mechanism is also proposed here. Finally, we demonstrate how this mechanism verifies each of the data property constraints.

4.1 Computing the buffers optimal size

The `data lifetime bound method` computes the upper bound of the number of times the writer can produce new values while a given data sample is still in use. On the other hand, the largest amount of time it can take for a job of a given task corresponds to the task worst-case response time. For a preemptive scheduling, such a value corresponds to the execution

completion of the first job of each task when the latter is released simultaneously with all its higher priority tasks [5]. The access to shared buffers being subject to the `single writer many readers principle`, the amount of used time a data sample depends on the execution of competing tasks on the shared buffer. Subsequently, we calculate below the required size for each of the communication buffers for both the linear and forked relationships between tasks as defined in Section 3.3.2.

Theorem 1 (Linear relation) *We consider a communication resource ∇^* such that $\exists \mathcal{E}^* = \{\tau_i|\tau_j\}$. The optimal size of ∇^* denoted by $|\nabla^*|$ is given by*

$$|\nabla^*| = \left\lceil \frac{R_j}{T_i} \right\rceil \tag{4.1}$$

where $|\nabla^*|$ is the cardinality of ∇^*, T_i is the period of τ_i, T_j and R_j are the period and the worst case response time of τ_j, respectively.

Proof 1 *In this thesis we consider that the writing of a new data sample coincides with the execution completion of the task job that produced this data, where the access to the shared buffer is asynchronous non-blocking. Subsequently, in order to guarantees the protection of the data being used by a job of τ_j, all the data samples that may be produced at the execution completion of the job(s) of τ_i must not overwrite the data being used. The number of jobs of τ_i than can complete within R_j time unites is equal to $\left\lfloor \frac{R_j}{T_i} \right\rfloor$. To that number we add one in order to protect the slot containing the data being used.*

For a task consuming several different messages whose related data samples are stored in separated buffers, thus Equation 4.1 remains valid. Hence, for a set of tasks $\{\tau_{j,1}, \cdots, \tau_{j,k}\}$ producing, respectively, the set of messages $\{m_{j,1}, \cdots, m_{j,k}\}$ consumed by the task τ_j, the size of the corresponding buffers is computed as $|\nabla^{m_{j,i}}| = \left\lceil \frac{R_j}{T_{j,i}} \right\rceil$ provided $1 \leq i \leq k$ where k is the number of messages that τ_j consumes. The tasks being into the linear relation may have different periods.

If $T_{j_i} = T_j$, it means that τ_{j_i} and τ_j jobs are always released at a same time instant. Though, the data item produced by a job of τ_{j_i} is consumed by a job of τ_j before the next release of τ_{j_i}. If the idea is to protect a data read by a job of τ_j that could be overwritten by a job of any $\tau_{j_i}, \forall 1 \leq i \leq k$, if $priority(\tau_j) > priority(\tau_{j_i})$, the job of τ_{j_i} is able to start

its execution only when the one of τ_j has finished. Considering that in such a configuration the next job of τ_j is also the first to start its execution, we can conclude that a buffer size of one is sufficient since a job of τ_{j_i} never overwrites a data consumed by the one of τ_j. This is also valid if priority$(\tau_j) <$ priority(τ_{j_i}). If $T_{j_i} > T_j$, then a data sample produced by a job of τ_{j_i} can be read by several jobs of τ_j and a job of τ_{j_i} can execute only if there is no active job of τ_j considering that the executing job of τ_j always preempts the execution of a job of τ_{j_i}, if any, due to the high priority of τ_j over τ_{j_i}. Therefore, a buffer of size one is also sufficient. Finally, if $T_{j_i} < T_j$, then new data samples related to m_{j_i} are likely to be produced between the activation and the completion of a job of τ_j considering that the executing job of τ_{j_i} always preempts the execution of a job of τ_j, if any, due to the higher priority of τ_{j_i} over τ_j. Utmost, within R_j time units $\left\lceil \frac{R_j}{T_{j_i}} \right\rceil$ new data samples can be produced by τ_{j_i} .

Theorem 2 (Fork relation) *We consider a communication resource ∇^* such that $\mathcal{E}^* = \{\tau_i | \tau_{i_1}, \cdots, \tau_{i_k}\}$ where k is the number of tasks reading from ∇^* . The optimal size of ∇^* denoted by $|\nabla^*|$ is equal to the number of data samples that can be written by τ_i before the execution completion of a job of the slowest among all τ_{i_j} plus one.*

Hence,

$$|\nabla^*| = \max_{1<j\leq k} \left\{ \left\lceil \frac{R_{i_j}}{T_i} \right\rceil \right\} \tag{4.2}$$

where $|\nabla^*|$ is the cardinality of ∇_i, T_i is the period of τ_i and R_{i_j} the worst case response time of the task τ_{i_j}.

Proof 2 *When a given data sample is consumed by several tasks likely having different periods, the length of time window within which each of the consumer tasks requires the data sample may be different from one consumer to another. Thus, for the faster consumers this window is shorter compared to the slower consumers. Based on the data management applied to the circular buffer, at a time instant t, two tasks read of the data located in different slots of the buffer (as one of the advantages of such a buffer) or from the same buffer slot. Being that way, considering the case two or more tasks (possibly with different periods) are reading from the same slot, the buffer size computed by Equation 4.2 guarantees that the data used by the slowest among the consumers; which fits perfectly with the intention of Theorem 2. Of*

course, during this time window the quicker tasks read more than one data samples without worrying about a possible overwriting of a data sample in use. Hence, Equation 4.2 is correct.

Notice 7 (Case of multi-source messages) *Regarding the multi-source message, each buffer vector is considered as a separate buffer and managed based on the timing parameters of each of the message producers and the ones of this message consumers. Therefore, for the latter, the situation is as follows:*

For a communication resource ∇^* storing the data samples related to the message m such that $\mathcal{E}^* = \{\tau_{p,1}, \cdots, \tau_{p,k} | \tau_{c,1}, \cdots, \tau_{c,k}\}$

$$|\nabla^*_{p,i}| = \max_{1<j\leq k} \left\{ \left\lceil \frac{R_{c,j}}{T_{p,i}} \right\rceil \right\}, \forall 1 < i \leq l \tag{4.3}$$

where (p,i) is a given producer of the message m and l the total of them. Accordingly, (c,j) is a task among the ones consuming the data samples related to the message m. If $l = 1$, Equation 4.3 becomes the same as Equation 4.2.

From what precedes, in the remainder of this thesis, we use the term message to refer to any type of messages. Hence, the prefixes uni-/multi- are explicitly used if the proposed solution cannot hold for both categories.

We describe within Algorithm 1 how to calculate the size allocated to each of the corresponding data variables. This algorithm considers both single-source and multi-source messages. The steps and clarifications related to this algorithm are explained progressively within the rest of this chapter.

Algorithm 1 *The circular buffer data variable memory allocation*

Require $\mathcal{G} = (\mathcal{T}, \mathcal{M}, \mathcal{E})$ ▷ The communication graph $\mathcal{G}$ returns sets of tasks and messages as well as links between them.
Compute the worst-case response time, R_i, for each $\tau_i \in \mathcal{T}$ ▷ Required to compute and bound the buffer sizes.
Retrieve from $\mathcal{G}$ the set of message domains E
Split E into message domains $\mathcal{E}^1, \cdots, \mathcal{E}^{|\mathcal{M}|}$ ▷ Within E there are as many message domains as there messages in $\mathcal{M}$, $|\mathcal{M}|$.
for each $\mathcal{E}^m$ such that $1 \leq m \leq |M|$ **do** ▷ Each message m in $\mathcal{M}$ has its own message domain denoted by $\mathcal{E}^m$.
    int $size = 0$
    for each $v_k \in \mathcal{D}^m$ such that $1 \leq k \leq nV$ **do** ▷ $\forall m$, there $\exists$ a data variable $\mathcal{D}^m$ made of nV member variables.
        if type of v_k is *static* **then** ▷ Memory allocation size is a-priori known and never changes (int, float,$\cdots$).
            $size \leftarrow size + SizeOf(type)$
        else ▷ The required memory size depends on the value to be written (**char** for instance).
            Set the maximum length ($maxLength$) to be allocated ▷ For such types, set the maximum length to be allocated.
            $size \leftarrow size + maxLength$
        end if ▷ The value of **size** is the number of bytes requires to store a single data sample related to m.
        $size \leftarrow size + paddingAddr^1(\mathcal{D}^m)$
    end for
    Separate the sets of **producers** and **consumers** for the message m
    Compute the number of producers and consumers: $nP \leftarrow |producers|$ and $nC \leftarrow |consumers|$
    for each $\mathcal{E}^m(i)$ such that $1 \leq i \leq nP$ **do** ▷ Here i is one of the producers for m and writes into a separate buffer, $\nabla^m(i)$
        if $nC = 1$ **then**
            $|\nabla^m(i)| = \left\lceil \frac{R_{consumers[0]}}{T_{producers[i-1]}} \right\rceil$ ▷ Equation 4.1
        else
            $|\nabla^m(i)| = \max_{1<c\leq nC} \left\{ \left\lceil \frac{R_{consumers[c]}}{T_{producers[i-1]}} \right\rceil \right\}$ ▷ Equation 4.2
        end if ▷ Here $|\nabla^m(i)|$ is the maximum number of data samples that can be hold into the corresponding $\mathcal{D}^m$
        Allocate ($size \times |\nabla^m(i)|$) bytes for $\mathcal{D}^m$ ▷ The data variable occupies $size \times |\nabla^m(i)|$ contiguous bytes
    end for
end for

4.2 The sub-sampling rate mechanism

The circular buffer offers many advantages, including the ability of different tasks to read from different locations of the buffer at a same time instant. The reason for this lies in the fact that each consumer task accesses the data stored in the buffer simply by selecting the data being into the slot pointed by the value of its own tail. This ability makes it much more suitable for use in systems where it may be necessary to use all the data produced (no sample data should be overwritten/lost before use). Each consumer has the possibility to consume the data, one by one, at its own pace. Such an implementation of the circular buffer is integrated in the RTMaps [2] [68] environment tool. That being, for the pairs of producer and consumer tasks with different periods, such an implementation would require the use of unbounded buffers while leading to the data aging before use. For embedded real-time systems where the data are subject to constraints such as data freshness, this operating mode

[2] RTMaps^{TM} stands for Real-Time Multisensor Applications, it is a highly-optimized component-based development and execution software tool. Thanks to RTMaps™ you can design, develop, test, benchmark and validate multisensor applications for Advanced Driver Assistance Systems (ADAS) and Highly Automated Driving (HAD) software functions but also advanced features in other domains such as autonomous and mobile robotics, energy, system monitoring, complex instrumentation and human factors.

may not implicitly meet the timing requirements applied to these data. This can be seen as a weakness in terms of meeting timing constraints imposed on the data. In this thesis, we exploit the best of what the circular buffer management may offer with several adaptations for a reliable and efficient asynchronous access to communication buffers. The goal is to have sufficient flexibility so as to preserve all the data properties under consideration. From this perspective, we propose a mechanism defining the communication buffer access rules, referred to as `sub-sampling rate mechanism`. We provide below its description.

4.2.1 Introduction

Before going more into details regarding the functioning of this mechanism, let us consider an example of three tasks $\tau_1(1,3)$, $\tau_2(2,8)$ and $\tau_3(3,12)$ communicating through a shared communication resource ∇^* such that $\mathcal{E}^* = \{\tau_1|\tau_2,\tau_3\}$. These tasks have implicit deadlines and are scheduled in accordance with the assumptions considered in this thesis. The first value within the pair of temporal parameters of a task is its worst-case execution time, while the second is the task period. The worst-case response time for τ_i, denoted by R_i are $R_2 = 3$ and, respectively, $R_3 = 8$. From Equation 4.2, the size of ∇^* is computed as follows: $|\nabla^*| = \max\left\{\left\lceil\frac{R_2}{T_1}\right\rceil, \left\lceil\frac{R_3}{T_1}\right\rceil\right\} = \max\left\{\left\lceil\frac{3}{3}\right\rceil, \left\lceil\frac{8}{3}\right\rceil\right\} = 3\,slots$. In Figure 4.3 we present the scheduling results for these tasks and highlights the expected results with regard to the sub-sampling rate mechanism.

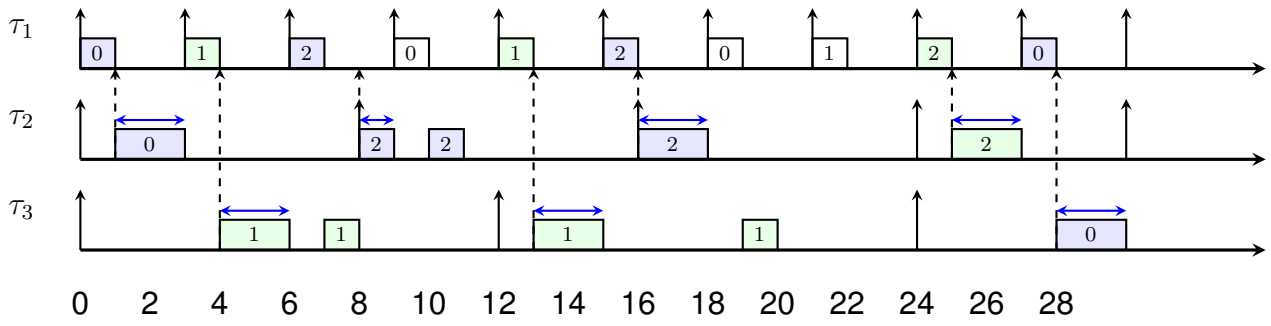

Figure 4.3: *The sub-sampling rate mechanism expected results*

The buffer slots are numbered 0, 1 and 2. The task τ_1 writes in these slots, circularly from the slot number 0 to the slot number 2 and loops back each time the maximum length

is reached. Likewise, each of τ_2 and τ_3 accesses the data from slot to slot. In this figure, the index of the slot into/from which the writing or reading operation took place is shown within the boxes indicating the execution window of the task. The blue lines above the boxes indicate the amount of time elapsed between the activation of the consumer task job and the release of the next job of the producer task. The down to upstream dotted arrows point to the execution completion of the producer task job generating the data sample consumed by the pointing job of the consumer task.

Basically, the purpose behind the sub-sampling mechanism is to ensure that at the consumer job activation time, this job ignores all the data samples produced between the previous and the current activation with respect to a given consumer task. So, for a consumer task τ_i, we denote by $\eta^p_{i,p-1}$, the number of data samples produced within the time window from the activation of the $(p-1)^{th}$ to the $(p)^{th}$ jobs of τ_i. As for instance, considering the example presented in Figure 4.3, we have $\eta^2_{2,1} = 2$, $\eta^3_{2,2} = 3$, $\eta^4_{2,3} = 3$, $\eta^2_{3,1} = 3$ and $\eta^3_{3,2} = 5$. The value of $\eta^p_{i,p-1}$ is likely to be different from one activation to another. The sub-sampling rate mechanism ensures that the jobs of the consumer tasks always access the most recently written data sample by jumping to the $\left(\eta^p_{i,p-1}\right)^{th}$ one.

4.2.2 Formalization and implementation

We consider a communication resource ∇^* such that $\mathcal{E}^* = \{\tau_i | \tau_{i_1}, \cdots, \tau_{i_k}\}$ where k is the number of consumers. For a task $\tau_{i_j} \in \{\tau_{i_1}, \cdots, \tau_{i_k}\}$, its $(p)^{th}$ job is activated at a time $Start(p, i_j)$ such that

$$Start(p, i_j) = Release(p, i_j) + \mathcal{I}^p_{i_j,h} \tag{4.4}$$

with

$$Release(p, i_j) = (p\text{-}1) \times T_{i_j} \tag{4.5}$$

$$\mathcal{I}^p_{i_j,h} = \sum_{h \in hp(i_j)}^{Start(p,i_j)} \left\lceil \frac{\Delta T(p, i_j)}{T_h} \right\rceil \times C^p_h \tag{4.6}$$

and

$$\Delta T(p, i_j) = Start(p, i_j)\text{-}Release(p, i_j) \tag{4.7}$$

where

- $hp(i_j)$ is the set of higher priority tasks than τ_{i_j} executed within the time interval between the release time of the $(p)^{th}$ job of τ_{i_j} and its activation time $Start(p, i_j)$, denoted by $\Delta T(p, i_j)$.
- $\mathcal{I}^{p}_{i_j,h}$ is the interference induced by all the $hp(i_j)$ tasks over τ_{i_j} within $\Delta T(p, i_j)$.
- C^{p}_{h} is the exact execution duration of each of the $\left\lceil \frac{\Delta T(p,i_j)}{T_h} \right\rceil$ jobs of τ_h that may have executed within $\Delta T(p, i_j)$ such that $c_h \leq C^{p}_{h} \leq C_h$ where c_h and C_h are the best- and worst-case execution times for each τ_h.

Notice 8 *In this thesis, it is not necessary to know how long it takes for a job to be completed. Instead, we set up a control mechanism allowing to condition the execution completion of a job by a successful writing of the data into all output buffers.*

For the same $\tau_{i_j} \in \{\tau_{i_1}, \cdots, \tau_{i_k}\}$, the activation time of the previous job, the $(p\text{-}1)^{th}$, is denoted by $Start(p\text{-}1, i_j)$ and calculated analogically as

$$Start(p\text{-}1, i_j) = Release(p\text{-}1, i_j) + \mathcal{I}^{p\text{-}1}_{i_j,h} \tag{4.8}$$

with

$$Release(p\text{-}1, i_j) = (p\text{-}2) \times T_{i_j} \tag{4.9}$$

$$\mathcal{I}^{p\text{-}1}_{i_j,h} = \sum_{h \in hp(i_j)}^{Start(p\text{-}1,i_j)}{}_{Release(p\text{-}1,i_j)} \left\lceil \frac{\Delta T(p\text{-}1, i_j)}{T_h} \right\rceil \times C^{p\text{-}1}_{h} \tag{4.10}$$

and

$$\Delta T(p\text{-}1, i_j) = Start(p\text{-}1, i_j)\text{-}Release(p\text{-}1, i_j) \tag{4.11}$$

Analyzing from the producer task side, we consider that the data sample consumed by the $(p\text{-}1)^{th}$ job of τ_{i_j} is produced by the q^{th} job of τ_i at a time instant denoted by $End(q, i)$

and calculated as

$$End(q,i) = \left\lfloor \frac{End(p\text{-}1,i_j)}{T_i} \right\rfloor \times T_i + C_i^q + \mathcal{I}_{i,h}^q \tag{4.12}$$

with

$$\mathcal{I}_{i,h}^q = \sum_{\substack{Release(q,i) \\ h \in hp(i)}}^{End(q,i)} \left\lceil \frac{\Delta T(q,i)}{T_h} \right\rceil \times C_h^q \tag{4.13}$$

and

$$\Delta T(q,i) = End(q,i)\text{-}Release(q,i) \tag{4.14}$$

We now calculate the time elapsed from the activation of the $(p\text{-}1)^{th}$ job of τ_{i_j} (activated at $Start(p\text{-}1,i_j)$; the one that consumed the data sample produced by the q^{th} job of τ_i which completed its execution at the time instant $End(q,i)$) to the release of the $(q+1)^{th}$ job of τ_i occurred at a time $Release(q+1,i)$. The resulting time delay is denoted by $delay_{Start(p\text{-}1,i_j)}^{Release(q+1,i)}$ and computed as

$$delay_{Start(p\text{-}1,i_j)}^{Release(q+1,i)} = Release(q+1,i)\text{-}Start(p\text{-}1,i_j) \tag{4.15}$$

Such a delay is represented in Figure 4.3 by the blue line over the job execution window box. When the activation of the $(p\text{-}1)^{th}$ job of τ_{i_j} happens immediately after the execution completion of the q^{th} job of τ_i, then we have $delay_{Start(p\text{-}1,i_j)}^{Release(q+1,i)} = Release(q+1,i)\text{-}Start(q,i)$. For instance, using the same model, we have $delay_{Start(1,2)}^{Release(2,1)} = 2$, $delay_{Start(2,2)}^{Release(4,1)} = 1, \cdots$.

Further, the delay separating the release time of the $(q+1)^{th}$ job of τ_i and the activation time of the p^{th} job of a task $\tau_{i_j} \in \{\tau_{i_1}, \cdots, \tau_{i_k}\}$ is computed as follows

$$delay_{Release(q+1,i)}^{End(p,i_j)} = End(p,i_j)\text{-}\left(End(p\text{-}1,i_j) + delay_{End(p\text{-}1,i_j)}^{Release(q+1,i)}\right) \tag{4.16}$$

Since the data consumed by the $(p\text{-}1)^{th}$ job of τ_{i_j} is produced during the $(q)^{th}$ period of τ_i and each task produces only one data per period, then $delay_{Release(q+1,i)}^{End(p,i_j)}$ is the effective time that τ_i can use to generate new data samples from the release of its $(q+1)^{th}$ job until the activation of the $(p)^{th}$ job of τ_{i_j}. Accordingly, we formulate Theorem 3.

Theorem 3 *We consider a communication resource ∇^* such that $\exists \mathcal{E}^* = \{\tau_i | \tau_{i_1}, \cdots, \tau_{i_k}\}$, where k is the number of consumer tasks. The number of data samples written into ∇^* by τ_i between the activation of the $(p\text{-}1)^{th}$ and p^{th} jobs of τ_{i_k}, denoted by $\eta^p_{i_j,p\text{-}1}$, is given*

$$\eta^p_{i_j,p\text{-}1} = \left\lceil \frac{delay^{End(p,i_j)}_{Release(q+1,i)}}{T_i} \right\rceil \tag{4.17}$$

Proof 3 *Referring to the assumptions considered in this thesis, each job starts reading when it enters the run state and continues to read the same data until it finishes. If this job happens to be preempted before it finishes reading from all the input buffers (there may be more than one input buffers), then when it resumes, it starts from where it was preempted. Regarding the preemption, if a task of higher priority enters the running state, it preempts the one of lower priority and executes until the completion if there is no other higher priority task released in the meanwhile. Considering also that the job execution completion coincides with the end of the data writing process, it is obvious that, within the $delay^{End(p,i_j)}_{Release(q+1,i)}$, the producer task has the capability of writing $\eta^p_{i_j,p\text{-}1}$ data samples, provided that the tasks are released periodically. Hence, Theorem 3 is correct.*

For instance, we illustrate the situation presented in Theorem 3 by considering the scheduling results presented in Figure 4.3. We obtain the following results:

$$\begin{cases} \eta^2_{2,1} = \lceil \frac{(1\times 8+0)\text{-}(0\times 8+1+2)}{3} \rceil = \lceil \frac{5}{3} \rceil = 2 \\ \eta^3_{2,2} = \lceil \frac{(2\times 8+0)\text{-}(1\times 8+0+1)}{3} \rceil = \lceil \frac{9}{3} \rceil = 3 \\ \eta^4_{2,3} = \lceil \frac{(3\times 8+1)\text{-}(2\times 8+0+2)}{3} \rceil = \lceil \frac{7}{3} \rceil = 3 \\ \eta^2_{3,1} = \lceil \frac{(1\times 12+1)\text{-}(0\times 12+4+2)}{3} \rceil = \lceil \frac{7}{3} \rceil = 3 \\ \eta^3_{3,2} = \lceil \frac{(2\times 12+4)\text{-}(1\times 12+1+2)}{3} \rceil = \lceil \frac{15}{3} \rceil = 5 \end{cases} \tag{4.18}$$

Starting from the results of Lemma 3, we identify in Theorem 4 which buffer slot the $(\eta^p_{p\text{-}1})^{th}$ data is written.

Theorem 4 *We consider a communication resource ∇^* such that $\exists \mathcal{E}^* = \{\tau_i | \tau_{i_1}, \cdots, \tau_{i_k}\}$ where k is the number of consumer tasks and $\eta^p_{i_j,p\text{-}1}$ is the number of generated data samples*

between the activation of the $(p\text{-}1)^{th}$ *and the* p^{th} *jobs of* τ_{i_j}*. The* $\left(\eta^{p}_{i_j,p\text{-}1}\right)^{th}$ *data sample is written into the slot number given by*

$$\left(\eta^{p}_{i_j,p\text{-}1} + tail\,(i_j, \nabla^*)^{p\text{-}1}\right) \mod |\nabla^*| \tag{4.19}$$

where $tail\,(i_j, \nabla^*)^{p\text{-}1}$ *is the index of the slot of* ∇^* *from where the* $(p\text{-}1)^{th}$ *job of a task* τ_{i_j} *read.*

Proof 4 *The proof of this theorem is based on the organization of the circular buffer itself. Indeed, such a buffer has a fixed size and the writing of data is organized in such a way that the data writing loops back to* 0 *after the slot with the maximal index is reached. Thus, knowing the number of data produced since the previous activation, the slot from which the reading of the* $(p\text{-}1)^{th}$ *job of* τ_{i_j} *took place and the size of the buffer, we sum them and apply the modulo operator to loop back in case* $\left(\eta^{p}_{i_j,p\text{-}1} + tail\,(i_j, \nabla^*)^{p\text{-}1}\right) > |\nabla^*|$*; which confirms the statement of Theorem 4.*

Until now, we didn't have any way to calculate this (slot) value because none of the consumers know the producer of the data they are consuming and therefore it looks like it is not possible to know exactly in which slot the $\left(\eta^{p}_{i_j,p\text{-}1}\right)^{th}$ data sample is located. Fortunately, even though the communication is isolated, the sharing of the resources is organized in such a way that the producer task has a full access to each of its output buffers and likewise do the consumer task regarding its input buffers as shown in Figure 3.11. That to say that the two buffer pointers (head and tail) remain accessible to the producer as well as to the consumer tasks. By using this opportunity, the problem can be solved.

Indeed, from Theorem 4 we obtain that the $\left(\eta^{p}_{i_j,p\text{-}1}\right)^{th}$ data sample is produced by the $\left(q + \eta^{p}_{i_j,p\text{-}1}\right)^{th}$ job of τ_i. The reason for this statement is that at the activation of the $(p\text{-}1)^{th}$ job of a task τ_{i_j}, q jobs of τ_i had completed their executions. On the other hand, the $\left(\eta^{p}_{i_j,p\text{-}1}\right)^{th}$ data sample is the recently produced by τ_i before the activation of the p^{th} job of τ_{i_j}. Accordingly, since the value of the `head` points to the slot where to write at the completion of the next job of τ_i, we can conclude that the $\left(q + \eta^{p}_{i_j,p\text{-}1}\right)^{th}$ is located into the slot of index given by $head(\nabla^*)\text{-}1$. However, in order to avoid the negative index values, the

$\left(q + \eta^{p}_{i_j,p\text{-}1}\right)^{th}$ data sample is located into the slot index given by

$$slotIndex = (\text{-}1 + head(\nabla^*) + |\nabla^*|) \mod |\nabla^*| \tag{4.20}$$

where $|\nabla^*|$ is the size of the communication resource ∇^*. In conclusion, for this mechanism, the objective is that each time a job of a given consumer task is activated, it must read the data sample being into the buffer slot pointed by the result returned by Equation 4.20. We implement this mechanism within Algorithm 2 and we add comments to each statement in order to ease its understanding.

Algorithm 2 *Implementation of the sub-sampling rate mechanism*

1: **return** the higher priority task τ_{prior}, its input buffers $\mathcal{B}^{In}_{prior}$ and its output buffers $\mathcal{B}^{Out}_{prior}$
2: **switch** ϖ_{prior} **do** ▷ Check the task communication state
3: **case** -1 ▷ $\varpi_{prior} =$ -1 if the job of *prior* has not yet finished reading all input data or if it leaves the sleeping mode.
4: **for** each $\nabla(i) \in \mathcal{B}^{In}_{prior}$ **do** ▷ The data variable $\mathcal{V}$ of each ∇ comprises i array vectors, numbered from 0 to i-1.
5: **for** j=0 To i-1 **do** ▷ The **readEnd** boolean is *prior* have read from the j^{th} vector at-least once.
6: **if** readEnd $= False$ **then** ▷ If it's the 1^{st} reading, *prior* points to the slot holding the freshly written value.
7: $tail \leftarrow$(getHead($\nabla(j)$)-1+|$\nabla(j)$|) $\mod |\nabla(j)|$ ▷ Equation 4.20
8: **end if** ▷ If not, *prior* points to the slot used during the first reading round.
9: readFrom$(prior, \nabla(j), \mathcal{V}(j), tail)$ ▷ In either cases, prior reads from the slot pointed by the value of tail
10: **if** readEnd $\neq True$ **then** ▷ After successful reading, set the slot to be read from during this execution
11: setTail$(prior, \nabla(j), (1 + tail + |\nabla(j)|) \mod |\nabla(j)|)$
12: readEnd $\leftarrow True$ ▷ Mention the fact that reading has taken place for at-least one time.
13: **end if**
14: **end for**
15: **end for**
16: $\varpi_{prior} \leftarrow 0$ ▷ Mention the fact that *prior* has finished to read from all buffers from $\mathcal{B}^{In}_{prior}$.
17: **case** 0
18: Process using the read data ▷ In the meanwhile, if the reading program is requested, return to Statements 3-16.
19: **if** $procEnd = True$ **then** ▷ Once processing is completed, output data must be written; no more reading.
20: $\varpi_{prior} \leftarrow 1$
21: **end if**
22: **case** 1 ▷ *prior* enters the writing stage and must read from all the output buffers.
23: **for** each $\nabla \in \mathcal{B}^{Out}_{prior}$ **do**
24: $head \leftarrow$getHead(∇) ▷ **head** has the value of the buffer slot to be written in with respect to ∇
25: writeIn$(prior, \nabla, \mathcal{V}, head)$ ▷ *prior* performs the writing into the buffer slot pointed by **head**
26: setHead$(prior, \nabla, (1 + head) \mod |\nabla|)$ ▷ Set new value of **head** to be used at the completion.
27: **end for**
28: $\varpi_{prior} \leftarrow$ -1 ▷ After writing completion, *prior* enters sleeping mode
29: $procEnd \leftarrow False$ ▷ Reset to **false** the variable indicating the end of the processing stage.

4.3 On the verification of local consistency constraints

In this section we present how the sub-sampling rate mechanism deals with each of the local consistency constraints while considering an asynchronous non-blocking access to the shared buffer. Precisely, as described in Section 2.3.3.2 the local consistency constraint **LocalCC**$\langle \tau_i, r, \tau_j \rangle$ is verified if, all the data samples written by τ_i into r, the jobs of τ_j always read data samples that are consistent, fresh and locally coherent. Precisely, a data sample is said to be `consistent` if it has the property 1, `fresh` if it has the property 2 and `locally coherent` if it has the property 3.

4.3.1 Data freshness

The freshness property (Property 2) constraint requires that the jobs of the consumer tasks for a given message should always consume the recently written data sample. The question here is *"does the sub-sampling rate mechanism ensures the data local freshness properties?"*.

In order to verify this constraint, we consider a communication resource ∇^* such that $\exists \mathcal{E}^* = \{\tau_i | \tau_{i_1}, \cdots, \tau_{i_k}\}$ where τ_i is the producer task and τ_{i_j} is a given consumer task provided $1 \leq j \leq k$.

Considering the activation time of the $(p-1)^{th}$ and $(p)^{th}$ jobs of τ_{i_j} such that the $(p-1)^{th}$ job consumes the data produced by the q^{th} job of τ_i. Thanks to Equation 4.17 (resulting from the Theorem 3) we obtain the number of data samples that can be written by the producer task in a time interval between the activation of the $(p-1)^{th}$ and the p^{th} jobs of any τ_{i_j}, denoted by $\eta^p_{i_j,p-1}$. In accordance with the sub-sampling rate mechanism, when the $(p)^{th}$ job of τ_{i_j} is activated, it consumes the $\left(\eta^p_{i_j,p-1}\right)^{th}$ data sample produced by τ_i since the activation of the $(p-1)^{th}$ job of τ_{i_j}; which is the recently produced data sample at this date. So, we conclude that implementing the `sub-sampling rate mechanism` maintains the `data freshness` property (Property 2).

4.3.2 Data local coherence

Considering the Specification 2, the `data local coherence property` is ensured if, for the data samples related to a `multi-source` message, the consumer tasks for this message always read, from the array vector where each producer writes, the data sample which is fresh (Property 2).

We can affirm that the sub-sampling mechanism validates this property on the basis of the following elements:

- The communication buffer is implemented by default in the form of a two-dimensional array where each producer task writes in its own vector. If there is only one producer, the array becomes an one-dimensional array. On the other hand, the algorithm implementing the sub-sampling mechanism handles such an eventuality by providing each consumer with the possibility to read from each of these vectors and the producers with the possibility to write into their own vectors.
- There is no possibility of overlapping data resulting from different execution steps of the producer(s). Also, each job is supposed to read its output data at its activation time ($Stat.6-8$) and, if it happens that a new reading is necessary, the job in progress continues to use (Skip $Stat.6-8$) and it goes directly to $Stat.9$ the data read at the very first moment of its execution ($Stat.7$)). By doing so, combining part of the old data sample with part of the new data sample is avoided.
- Finally, as aforementioned, this freshness property is maintained by our mechanism.

Based on the above, we obtain that the data local coherence property is maintained.

4.3.3 Data consistency

The data consistency property constraint requires that no data samples in use should ever be overwritten. We verify this hypothesis by deduction. First of all, for the data consistency to be violated, the read data must be overwritten before the consuming job finishes its execution. At worst, this is when the overwriting occurs before the end of a duration equal to the worst response time of the corresponding task. The size of the buffers is calculated by taking into account the worst-response time of the consuming task. These sizes ensure that while there is

a job of the consumer task in progress, the generated data does not overwrite the read data. From the foregoing, one can simply confirm that the data consistency property is preserved.

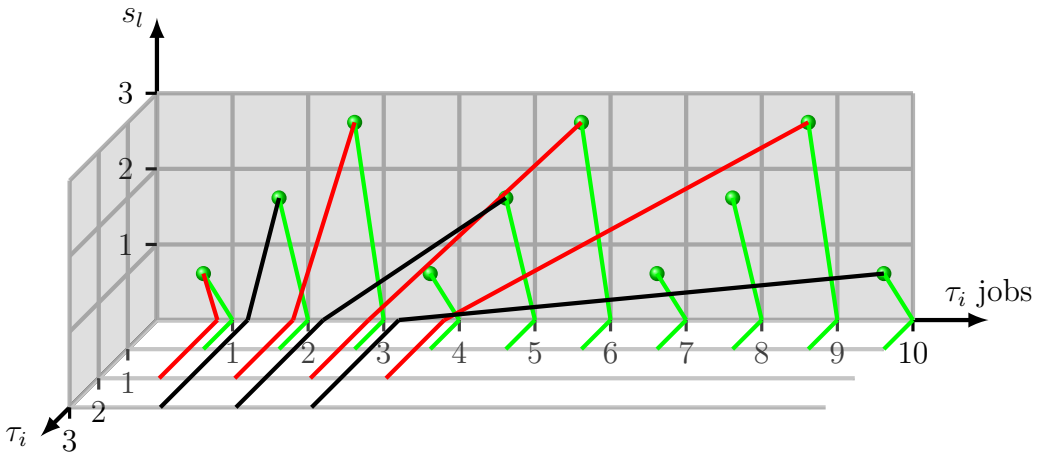

Figure 4.4: *An example showing the sub-sampling mechanism result*

To illustrate the above, let us consider a communication resource ∇^* such that $\mathcal{E}^* = \{\tau_1|\tau_2, \tau_3\}$ and $|\nabla^*| = 3$. The tasks have the following timing characteristics: $\tau_1(1, 3)$, $\tau_2(2, 8)$ and $\tau_3(3, 12)$. The first parameter is the worst-case execution time, the second is the period and all the tasks have implicit deadlines. The scheduling results are presented in Figure 4.3. The Figure 4.4 displays the slot into/from which the writing and reading operation of each of the jobs happened. The Z-axis represents the tasks from τ_1 to τ_3, while the Y-axis represents the slots of the buffer (there are three of them, each box represent a buffer slot). Finally, the X-axis represents the list of jobs (from one to ten jobs).

As the communication buffer has a storing capacity of 3 data samples, the data consistency is violated if and only if the τ_1 can manage to generate more than 3 data samples within the time interval bounded by the execution start and completion of a job of any of τ_2 or τ_3; which is not possible according to Equation 4.2. In Figure 4.4, the green lines point to the slots where each job of the producer task writes the output data samples represented by a circle green ball. The buffer slot are presented on the Y-axis and are numbered from 1 to 3. Herein, we can see that the new production of τ_1 never overwrote the data being used any of the jobs of τ_2 and τ_3, in red and black color lines, respectively.

Chapter

5 Global consistency constraints

Chapter content

5.1 CPU and memory storage requirements 93

5.2 Spindle modeling and formalization 94

5.3 The solution overview 99

5.4 Verifying the global consistency constraints 102

5.4.1 The last reader tags mechanism 102

5.4.2 Computing the spindle source buffer size 105

5.4.3 Spindle chains propagation delays 109

5.4.4 The scroll or overwrite mechanism 115

In this chapter, we present our solutions towards the verification of data properties global consistency constraints. Our solutions can be useful for wide variety of domains including data fusion for intelligent systems that infers the objects identity (or make some decisions) on the basis of the sensed data. When these data are generated by the same sensor, the part of the system (task) in charge of inferring (decision making) must be supplied with the capability to associate only the data initially resulting from the same execution step of the source (sensor) task. On the other hand, if the data to be combined are generated by different sources, only the ones acquired within the same time window on both sides reflect the same state of the environment. For example, autonomous vehicles (manned or unmanned), intelligent robots, etc. are equipped with multiple camera sensors that continuously interact with the surrounding environment. As aforementioned, an example of such application may

concern the identity inference of the object in the neighborhood (face, sides, ...). Depending on the collected data, a coherent decision must be taken with regard to the criticality of such a decision, i.e., the level of harm that may result from mistaken identification. For example, considering the autonomous vehicle, if the object in the neighborhood is another vehicle moving in the same direction, the decision would be to adapt the speed to the speed of the vehicle in front (accelerate or decelerate as appropriate) whereas as if the object in the neighbourhood is a crossing human, the decision should be to suddenly stop the vehicle.

It is important to note that in these circumstances, accuracy is an important consideration and all image sensors must synchronize or coordinate with each other. Nevertheless, the heterogeneity of these image sources (camera, lidar, radar,$\cdots$) increases the complexity of achieving an expected accuracy; they may have different delivery speeds, for instance. As a result, the collected data may not reach the image processing task at the same time. Subsequently, it must be ensured that the image processing task uses only data reflecting the same state of the environment.

In this thesis, sets of data reflecting the same state of the environment are referred to as `matching data`. We are concerned with data originating from same source, propagating through different paths to converge on a same task. Matching data are those that originally result from the same execution step of the data source task. Considering that the propagation of the data samples resulting from the same execution step of the source task may reach the associating task at different time instants, we aim to propose solutions/mechanisms allowing to postpone the reading of the data that arrived early to make them wait for those arriving later. These solutions have been addressed, previously, in the literature. For instance in [65, 66], the authors consider the assumption that the communication buffers are well configured to handle the proposed solution. Therefore, the proposed solution overlooks the impact of the communication system on the task system scheduling as well as on the maintenance of the properties of the data exchanged between adjacent tasks such as the data consistency property. In [68], the authors propose a synchronization mechanism associating the matching data in the FADE system [67]. In contrast, they consider unbounded buffers and the tasks execution order is not subjected to stringent real time scheduling requirements. The resulting system ("RTMaps") is a black box with no possibility to examine the source code compared

to PX4 which is an open source system. To the best of our knowledge, no solution considers both local and global constraints applied to the data properties.

5.1 CPU and memory storage requirements

There is a wide variety of mechanisms to address both local and global constraints applied to the data properties, but we need to estimate the costs they induce in terms of CPU and memory overheads or the possibility that it may have a negative impact on the initial task scheduling order or not. As for instance, the control of the disparities between ages is avoided by using arbitration mechanisms on shared variables. As previously mentioned, this type of solution is known to be potentially capable of perturbing the task scheduling order, which can lead to scheduling problems such as priority inversions, the deadlocks formation and, in extreme cases, to a chained blocking resulting in a non-schedulable system, especially when the considered system is complex like the one considered in this thesis. Nevertheless, despite the negative impact that this mechanism can have on the task scheduling order since it requires a high CPU overhead, it is still less greedier in terms of required memory space.

On the other hand, there are other mechanisms called asynchronous that allow each task to perform read/write operations when the scheduler allows it. This possibility of reading and writing without blocking can be achieved either `implicitly` or `explicitly`. For the former, each task is expected to have a local copy for each of the input variables it consumes. Afterwards, at the release time of each of its jobs, values from the global variables are copied to the corresponding local variables. These local variables are used by the current job whenever reading operation is requested. It should be noted that the use of such local copies is restricted to the area of adjacent tasks only. If the data propagation goes through intermediary tasks, this would require each intermediary consumer to make his own copies and so forth until the final destination. This mechanism doesn't seem to affect the job scheduling order but it may be extremely greedy in terms of memory space caused by the duplication of variables but also requires additional processor cycles for copying in and writing out from global back to local variables and from local to global variables. Regarding the `explicit asynchronous mechanisms`, the only cost to be paid is the memory overhead. Therefore, this mechanism is

preferable within our context, because we are concerned not to disrupt the task scheduling order while assuming that there is sufficient memory space.

Considering the benefits and drawbacks of both categories of mechanisms, we prefer the explicit asynchronous mechanisms and assume that there is sufficient memory space to store all the produced data in such a way that it is always guaranteed that, in spite of gaps in terms of delivery time, the consumer is able to select the temporally coherent data. In the remainder of this chapter we verify the global coherence and freshness for data originating from the same source. We call `"spindle"` the set of functional chains involved in the propagation of these data from source to destination form a network of chains and present below a more formal definition.

5.2 Spindle modeling and formalization

The `global matching constraint` concerns a task associating data originally produced by the same task and which propagated through different chains. Accordingly, the global matching constraint requires that only data samples resulting from the same execution step of the source application must be associated. Moreover, this requirement may not concern all the messages produced by the source task. As in [66], we use the term `spindle` to name the set of functional chains involved into the propagation of the data samples related to a given message from the source until the associating task.

Definition 14 (`Spindle`) *A spindle $\mathcal{S}_q^*(\tau_{src}, \tau_{last})$ is a set of q chains propagating the data samples related to the message * such that all these chains have the same* ***`source task`*** *denoted by τ_{src} and the same* ***`matching task`*** *τ_{last} task, with $q \geq 2$.*

Each chain in $\mathcal{S}_q^*$ is denoted by $\mathcal{C}_{c:q}^*$ with c the index of a specific chain, $\forall 1 \leq c \leq q$. The chains composing the spindle can be linear or branched.

Definition 15 (`Linear chain`) *A chain $\mathcal{C}_{c:q}^* \in \mathcal{S}_q^*$ is said* ***`linear`*** *iff $\forall c, c\prime : 1 \leq c, c\prime \leq q \wedge c\prime \neq c$ we have $\mathcal{C}_{c:q}^* \cap \mathcal{C}_{c\prime:q}^* = \varnothing$.*

Definition 16 (`Branched chain`) *A chain $\mathcal{C}_{c:q}^* \in \mathcal{S}_q^*$ is said* ***`branched`*** *iff $\exists \mathcal{C}_{c\prime:q}^* \in \mathcal{S}_q^*$ such that $\mathcal{C}_{c:q}^* \cap \mathcal{C}_{c\prime:q}^* \neq \varnothing$ with $1 \leq c, c\prime \leq q \wedge c\prime \neq c$.*

In the presence of one or many branched chains, the number of initial chains may differ from the one of chains reaching the end of the spindle. Accordingly, we denote by p the number of initial chains while q keeps the meaning of the number of the chains reaching the spindle sink task. For instance, regarding the spindle in Figure 5.3, $p = 2$ while $q = 3$.

On this basis, we distinguish the following classes of spindles: `Balanced`, α-`Balanced` and unbalanced spindles.

Definition 17 (`Balanced spindle`) *$\mathcal{S}_q^*$ is a balanced spindle iff $p = q$ and all q chains are linear.*

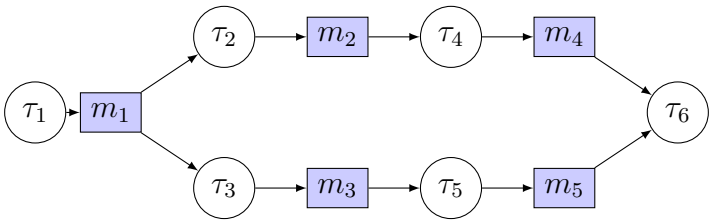

Figure 5.1: *A balanced spindle example.*

The example shown in Figure 5.1 is a balanced spindle since both its propagation chains are linear.

Definition 18 (α-`balanced spindle`) *$\mathcal{S}_q^*$ is a α-balanced spindle iff $p = q$ and there is a number α of embedded balanced spindles in its composition.*

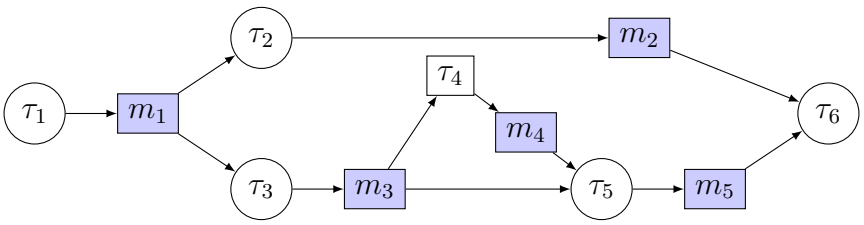

Figure 5.2: *An α-balanced spindle example.*

The spindle presented in Figure 5.2 has an inner balanced spindle regarding the message m_3 which propagates through the portions $\tau_3 \xrightarrow{m_3} \tau_4 \xrightarrow{m_4} \tau_5$ and $\tau_3 \xrightarrow{m_3} \tau_5$. Although for the spindle $\mathcal{S}_2^{m_1}$ $p = q$, having in its structure makes it an $\alpha-$balanced spindle.

Definition 19 (`Unbalanced spindle`) *$\mathcal{S}_q^*$ is an unbalanced spindle iff $p < q$.*

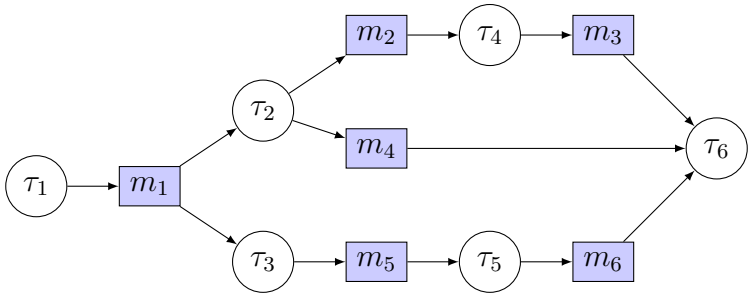

Figure 5.3: *An unbalanced spindle example.*

All the above types of spindles have in common the fact that they all propagate the data samples read from a same buffer and associated by a same task. Analogically, this source buffer is named `spindle source buffer` where p is the number of tasks reading from this buffer.

Notice 9 *In this thesis we consider only* ***`balanced`*** *and* ***`unbalanced`*** *spindles with no nested spindles. For the* ***`unbalanced`*** *spindle each* ***`branched chain`*** *is split into* ***`linear chains`*** *at the last common task.*

Definition 20 (`Spindle source buffer`) *The communication resource ∇^* is the spindle source buffer if there exists a spindle $\mathcal{S}_q^*$ such that $\mathcal{E}^* = \{\tau_{src}|\tau_1^{start}, \cdots, \tau_p^{start}\}$ provided $2 \leq p \leq q$.*

where τ_{src} is the producer into ∇^* and $\tau_1^{start}, \cdots, \tau_p^{start}$ are the corresponding consumer tasks. The expression $2 \leq p \leq q$ considers that the number of tasks reading from the spindle source buffer (p) may be less than the number of buffers from where the spindle sink task read (q) for the reasons aforementioned. On the other hand, the spindle sink task reads from q different buffers. Each of these buffers is written by the second to the last task in each chain. We name such buffers as the `spindle chain last buffer` that we denote by $\nabla^{*c}, \forall 1 \leq c \leq q$.

Definition 21 (`Spindle chain last buffer`) *$\nabla^{last_c^*}$ is a spindle chain last buffer there exists a message $last_c^*$ such that $\mathcal{E}^{last_c^*} = \{\tau_c^{last}|\tau_{sink}\}$ with $1 \leq c \leq p$.*

where τ_c^{last} is the second to last task belonging to the functional chain of index c. Here $\nabla^{last_c^*}$ is a message originally resulting from the propagation of a message m produced by the spindle source task τ_{src}.

Definition 22 (`Spindle inner buffers`) *For each spindle propagation chain $\mathcal{C}^*_{c:q}$ such that τ_c^{start} and τ_c^{last} are the second and the second-to-last tasks belonging to $\mathcal{C}^*_{c:q}$. Any buffer placed between τ_c^{start} and τ_c^{last} is refereed to as a* ***`spindle inner buffers`****.*

The goal of this section is twofold: (i) computing the optimal size for each of the categories of buffers involved in data propagation and (ii) proposing suitable data access mechanisms to correctly maintain the said properties. Specifically, we aim to ensure that the spindle sink task (τ_{sink}) always reads, from each of the spindle chain last buffers, the data samples resulting from the same execution step of the spindle source task (τ_{src}). This is based on a general model where some parts (or sub-systems) may be concerned by the use of matching data (involved in the spindle data propagation). The proposed mechanisms must provide ways and means for the individual tasks to implement a data access behaviour taking into account this aspect.

Discussing the sub-systems concerned with the spindle data propagation, the temporal characteristics of the tasks and the composition of the spindle chains have an important impact on the verification of global consistency constraints. For instance, the sequence of tasks involved in the spindle data propagation across a given chain may have different sampling periods. It is therefore possible to end up in so-called `under-sampling` pattern (the producer task writes data samples faster than the consumer reads them, $T_{producer} < T_{consumer}$) or `over-sampling` pattern (the consumer reads data samples faster than the producer writes them, $T_{producer} > T_{consumer}$). Regarding the `under-sampling pattern`, the direct consequence is that some data samples may be overwritten before being used, causing a rupture in data propagation before reaching the spindle sink task. As a result, the sink task may not find in each of the spindle chain last buffer the matching data samples produced during the same execution step of the spindle source task, unless these buffers have unbounded size. On the other hand, regarding the `over-sampling pattern`, the consequences may not be so fatal; each data sample written by the producer is ensured to continue its propagation (it is read at least once or possibly several times by the job(s) of the adjacent consumer task) which results in a no rupture of propagation. This being said, drawbacks may result from the `over-sampling pattern` as there may be several output data resulting from the same execution step of the spindle source task that may be buffered into a given spindle chain

last buffer; leading then to unwanted memory overhead. The other constraint is that all the spindle propagation chains may have different propagation delays for a data sample to propagate from the spindle source until the spindle sink task.

Considering all the above, the objective is to manage the propagation of the data from the source up to the spindle sink in such a way that when a job of the spindle sink task is activated, it finds in each of the spindle chain last buffers, the data samples originally resulting from the same execution step of the spindle source task and that, without using any type of synchronization mechanism as they may have a negative impact on the schedulability of the task system. In order to manage all the aforementioned particularities of the tasks system, we present below the stages of the proposed resolution plan to achieve the goal of this thesis with respect to the requirements and assumptions to be taken into account herein.

For a clear overview of the spindle model and related notations, we use the model shown in Figure 5.4 and major notations used in the spindle data propagation management are shown in Figure 5.5.

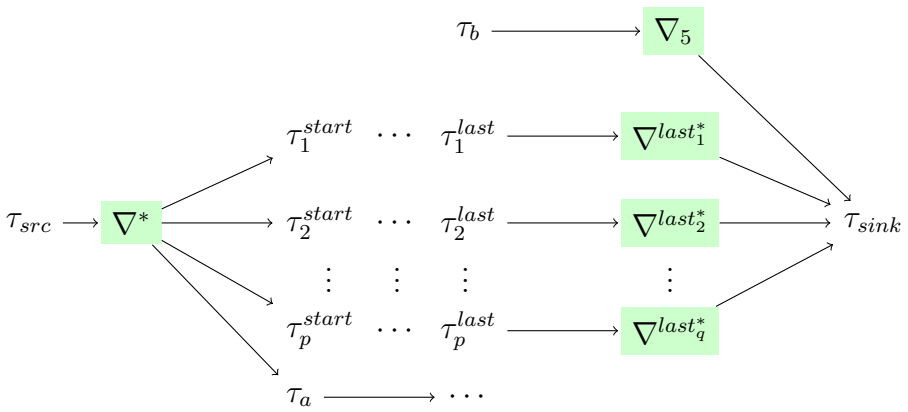

Figure 5.4: *A general configuration configuration of the spindle*

Symbol	Definition
τ_{src}	The spindle source task
τ_c^{start}	The second task belonging to the c^{th} spindle chain provided $1 \leq c \leq p$ where p is the number of tasks involved in the spindle data propagation which read from the spindle source buffer ∇^*.
$\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$	The lower priority tasks among the $\{\tau_c^{start}\}$, for all $1 \leq c \leq p$
τ_{sink}	The spindle sink task
τ_c^{last}	The second-to-last task belonging to the c^{th} spindle chain provided $1 \leq c \leq q$ where q is the number of chains outputting the data samples concerned by the spindle data propagation.
$\nabla^{last_c^*}$	The spindle chain last buffer belonging to the c^{th} spindle chain provided $1 \leq c \leq q$.
$\Omega_{c:q}^{min}$	The smallest amount of time it can take for a data sample read from the spindle source buffer to propagate until the output of the related data sample into the corresponding spindle source last buffer, with respect to the c^{th} spindle chain.
$\Omega_{c:q}^{max}$	The smallest amount of time it can take for a data sample read from the spindle source buffer to propagate until the output of the related data sample into the corresponding spindle source last buffer.

Figure 5.5: *Major notations used in the spindle data propagation*

5.3 The solution overview

We underline that the peculiarities due to the structure of the tasks system timing characteristics increase the complexity of verifying global consistency constraints especially where it requires that only matching data samples must be associated. The goal being to associate the data samples resulting from the same execution step of the spindle source task, there is no way to achieve it implicitly. Therefore, one must find a way to filter the data from the source task such that only the data samples that are likely to propagate until the sink task are considered from the spindle source buffer. If such a way exists, it would not only allow to find matching data samples in each of the spindle chain last buffers but also would

reduce the overload which might result from the propagation of several unnecessary data [1] from the source buffer. Nevertheless, the solution must not be limited solely to the objective of guaranteeing the data matching property; it must also have the ability of supporting the heterogeneous properties (data consistency and freshness for instance), required at different levels of the functional chain. We provide below the list of steps allowing to achieve this joint objective.

Step 1 *Force the jobs of each* τ_c^{start} *to propagate the same sample of data read from the spindle source buffer until the job of* $\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$*, which has read, finishes its execution. For a data sample reaching the end of a spindle chain, it increases the probability of finding matching data samples into other spindle chain last buffers. Thus, not only we decrease the size of the buffers at the end of each spindle chain, but also we avoid the propagation of data samples with a high probability of being preempted.*

Notice 10 *For the spindle source task* τ_{src} *and the tasks* $\tau_1^{start},\cdots,\tau_p^{start}$ *we consider the assumption that* $priority(\tau_{src}) > priority(\tau_c^{start})$ *with* $1 \leq c \leq p$ *and* p *the number of tasks involved in the spindle data propagation.*

It should be noted that not all tasks reading from the spindle source buffer are automatically part of the spindle chain; a functional chain is considered to be a spindle chain if it is involved in the spindle data propagation. Therefore, the solution resulting from Step 1 provides the necessary flexibility for other non concerned tasks. For instance, we consider the model presented in Figure 5.4 where ∇^* is a spindle source buffer such that $\mathcal{E}^* = \{\tau_{src}|\tau_1^{start},\cdots,\tau_p^{start},\tau_a\}$ where p is the number of tasks belonging to the functional chains involved in the spindle. Referring to the Step 1 statement, the tasks $\{\tau_1^{start},\cdots,\tau_p^{start}\}$ are mandated to always propagate the same data sample while the jobs of τ_a read the required input data on its own pace based on the τ_a scheduling priority. Given that these p tasks do not have the same periods, it is then obvious that the smaller period tasks are forced to propagate the same data many times until those proceeding slowly complete their executions. To that end, we propose in the Section 5.4.1, a mechanism called *"the last reader tags mechanism"*.

[1] Data samples which don't have the chance to propagate until the end of the corresponding chain due; rupture of propagation due to data overwriting

Step 2 *Computing the optimal (minimal) size of spindle source buffer such that the data consistency of the read data is always guaranteed.*

Given that we consider that access to shared buffers is done asynchronously without blocking, keeping a data read by one of the tasks until it is consumed by all the other tasks imposes to this data to stay in the buffer longer than usually; a sufficient size to store it before being overwritten must be computed. Such a size is computed in Section 5.4.2.

Step 3 *Computing the propagation delay for each of the spindle chains.*

Expecting to find corresponding data samples in each of the spindle chain last buffers requires to understand how long it can take for a data to propagate from the spindle source until the spindle sink if it is propagated through each of the spindle chains. Thus, in Section 5.4.3 we formally calculate the shortest and largest data propagation delay for each of the spindle chains.

Step 4 *Computing the optimal size of each of the spindle chain last buffer guaranteeing that a job of the spindle sink task will always find, in each of these buffers, at least one data sample resulting from the same execution step of the spindle source task.*

Using the results from Step 3, we can compute the size of each spindle chain last buffer. Given that the spindle sink task may read some data that are not concerned by the global matching property, it must be able to perform a correct reading depending on whether or not the input buffer belongs to a spindle chain. For instance, referring to the example in Figure 5.4, the computation of the size of each of the spindle chain last buffers $\nabla^{last_1^*}, \nabla^{last_1^*}, \cdots, \nabla^{last_q^*}$ follows the Step 4 statement whereas the one of ∇_5 is computed using Equation 4.1 considering the message domain $\mathcal{E}^* = \{\tau_b | \tau_{sink}\}$. The above sizes are calculated on the basis of worst-case propagation delays. So, they may be extremely large while containing useless duplicates. By "duplicates", we mean the data samples resulting from the same execution step of the spindle source task. To minimize these duplicates, leading to the minimization of the spindle chain last buffer sizes, we introduce a novel mechanism, referred to as `scroll or overwrite mechanism` regulating the data writing by each second-to-last task with respect to each spindle data propagation chain.

We present in Figure 5.6 a recapitulating diagram indicating which mechanism to apply at each level of the spindle propagation chain.

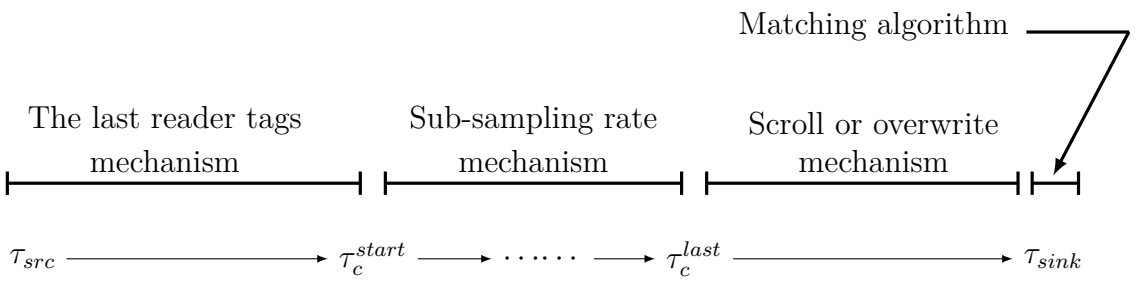

Figure 5.6: *The global consistency constraints verification scheme*

Precisely, for each second task τ_c^{start} involved in spindle data propagation and belonging to a chain of index c such that $\mathcal{E}^* = \{\tau_{src}|\tau_1^{start}, \cdots, \tau_p^{start}\}$ provided $1 \leq c \leq p$ where p is the number of such tasks, we apply the `last reader tags` mechanism. Then, regarding a spindle chain of index c, the data management for the tasks falling between its second and its second-to-last tasks, we apply the so-called `sub-sampling` mechanism developed in Section 4.2. Lastly, for each second-to-last task of each chain, the `scroll or overwrite` mechanism is applied to minimize the waiting buffers at the end of each chain.

5.4 Verifying the global consistency constraints

5.4.1 The last reader tags mechanism

The main idea behind the last reader tags mechanism resides in modifying the rules under which the access to the spindle source buffer is done. Generally the data reading is carried out such that each consumer asynchronously reads the data by means of his tail value, making it independent from the reading of another task. This property is acquired by considering the execution of the producer and the consumers regarding a given message, which are organized under the so-called message domain. When a job of the producer task, which must be the only one writing into a buffer intended to store the data related to a specific message, finishes its execution, it increments the head value which will be used to identify into which buffer slot its next job is going to write the output data. As for the individual consumer, on activation

date of each of its jobs, the latter consumes the data located into the buffer slot pointed out by the current value of tail. Only at the reading completion from each input buffer, the corresponding tail value is incremented to target the slot where the next job starts reading the data at its activation date. Using this operating approach (presented in Section 4.2) we ensure that the data constraints imposed at the local level (task-level data properties constraints) are guaranteed with the help of the sub-sampling mechanism, which is relevant only for inner buffers (not the source buffer or spindle chain last buffers).

For tasks released simultaneously, let us consider the following lemma, which regards exclusively the data sample read by the first job of each of the consumer tasks that may be reading from a given buffer said ∇^*.

Lemma 1 *When tasks are released simultaneously, if a data sample must be consumed by all the consumer tasks, then among the first jobs of each of the consumers, the job of the lower priority is the last one to use the first read data sample.*

Proof 5 *We consider that the reading operation can be performed at any time instant within the current job execution window, but the data to be read have to be the ones used at the activation time. Therefore, we can say that a job can use a data for at most a duration equal to its task worst-case response time, denoted by R_i. Further, given that a job is scheduled based on the task priority, then the lower is its priority, the later the job is scheduled, and subsequently, the longer this job can use the first data sample.*

This is observable considering the example presented in Figure 5.7, where four tasks τ_1, τ_2, τ_3 and τ_4, scheduled preemptively based on a fixed priority scheduling algorithm (rate monotonic algorithm [2]). The tasks have implicit deadlines and are characterized by the worst-case execution time and the period where their timing parameters are $(1, 6)$, $(1, 8)$, $(2, 12)$ and $(3, 18)$, respectively. The corresponding message domain is $\mathcal{E}^* = \{\tau_1|\tau_2, \tau_3, \tau_4\}$. The dotted arrows from τ_1 downstream to τ_2, τ_3 and τ_4 show which job read the first data produced by τ_1. Here the first job of τ_4 completes its execution lastly at $t = 8$.

Based on Lemma 1 and in order to enforce all the consumers reading from the spindle source buffer to propagate the same data sample, we need that all the concerned tasks use the same data as long as the job of the lower priority among all these consumers has not

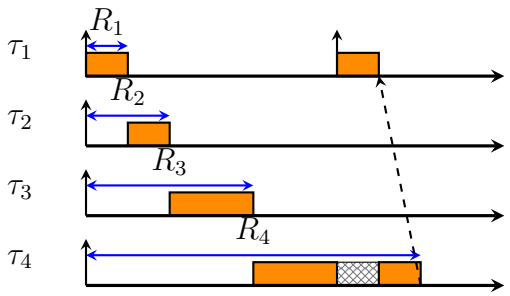

Figure 5.7: *Example showing the data tagging operation*

yet completed its execution. For instance, considering the example presented in Figure 5.7, each job of τ_2 and τ_3 that may be triggered at a time instant t such that $0 \leq t \leq R_4$ must read the data sample produced by the first instance of τ_1 at $t = 1$. Once the job of the lower priority task among the consumers completes, a new data to be consumed further has to be determined and must be that recently written by the spindle source task. The activity of determining which data sample is now going to be consumed from the spindle source buffer is called `data tagging` and, therefore, the data itself is called `tagged data` and the task determining this data is the `tagging task`; which explains the naming of this mechanism as `the last reader tags`.

Precisely, the tagging task is the lower priority task among the consumers reading from the spindle source buffer; τ_4 in the above example. The dotted arrow from the execution completion time of the first job of τ_4 upstream to τ_1 shows which data sample must be consumed by the jobs τ_2, τ_3 and τ_4 that may execute within time interval being between this completion time and the completion time of the next job of τ_4.

To generalize the results of Lemma 1 to all the data samples produced by the spindle source task, we consider a communication resource ∇^* holding the data samples related to the message $*$ such that ∇^* is a spindle source buffer. The corresponding message domain is $\mathcal{E}^* = \{\tau_{src} | \tau_1^{start}, \cdots, \tau_p^{start}, \tau_1, \cdots, \tau_n\}$ where $p+n$ tasks reading from ∇^* among which only p tasks are involved in the spindle data propagation operation. In Theorem 5 we provide this generalization.

Theorem 5 *We consider a message domain* $\mathcal{E}^* = \{\tau_{src} | \tau_1, \cdots, \tau_n, \tau_1^{start}, \cdots, \tau_p^{start}\}$ *such that* ∇^* *is a spindle source buffer where there are* p *tasks concerned by spindle data propagation and* n *tasks are not concerned.* $\forall i \in [1, n] \wedge c \in [1, p]$*, the data sample to be read is located*

into the slot pointed out by the following values of tail

$$\text{tail}(\tau_i, \nabla^*) \leftarrow (head(\nabla^*) - 1 + |\nabla^*|) \mod |\nabla^*| \tag{5.1}$$

and

$$\text{tail}(\tau_c^{start}, \nabla^*) \leftarrow tail\left(\underset{1 \leq c \leq p}{\text{LP}} \left\{\tau_c^{start}\right\}, \nabla^*\right) \tag{5.2}$$

Proof 6 *As previously mentioned, any type of task can be read from the buffer. For tasks which are not part of the spindle, the case is addressed the same as in the previous chapter. Here only the properties required at the local level are important. For this category of tasks, the result of Equation 4.20 which is the same as those described by Equation 5.1.*

As for tasks that are part of the spindle, they will only use data that has been tagged at completion of the previous job of lower priority tasks among them. This data must be the most recently produced by the spindle source task; which is also computed by Equation 4.20 considering only this lower priority task.

In Algorithm 3 we present a possible implementation of the `last reader tags mechanism`. In other words, it basically gives the main steps and the logic behind its implementation process. An effective implementation is indeed possible in compliance with specific programming language specifications.

5.4.2 Computing the spindle source buffer size

The implementation of the `last reader tags mechanism` imposes that all the jobs of the tasks that are part of the spindle data propagation keep reading the same data sample during the time interval separating the completion time of two consecutive jobs of the lower priority task among them. To ensure data consistency, there must be enough slots to hold new data samples produced during this time interval otherwise the data sample being processed may be overwritten before the completion of next job of this lower priority task. Therefore, it is necessary to first compute the largest amount of time a tagged data can stay stored within the source buffer before being overwritten. Such a delay is referred to as the *spindle source buffer data consistent interval* which we denote by `SCI`, computed based on the `data lifetime bound method`.

Algorithm 3 *Implementation of the last reader tags mechanism*

function MSGSPT(msgDomain) ▷ Takes a message domain as argument.
  return spindleTasks ▷ Returns the set of tasks involved in the spindle data propagation.
end function
function LOWERP(setOfTasks) ▷ Takes the set of tasks as argument.
  return LP ▷ Returns the lower priority among them.
end function

---

return the higher priority task *prior*, its input buffers ∇^{In}_{prior} and its output buffers ∇^{Out}_{prior}
Task lp ▷ Declare a task lp.
Buffer bf ▷ Declare a buffer bf.
if $\varpi_{prior} = -1$ **then** ▷ If the job of *prior* has not yet finished reading all input data or if it leaves the sleeping mode.
  for each $\nabla^m \in \nabla^{In}_{prior}$ such that $1 \leq m \leq |\nabla^{In}_{prior}|$ **do**
    if (∇^m) is not a spindle source buffer or $prior \notin msgSPT(\mathcal{E}^m)$ **then**
      Follow Algorithm 2 ▷ If the current buffer isn't the spindle source one or if *prior* is not involved in the spindle data propagation. A task may read from spindle source buffer without being in the spindle data propagation. In such cases, the Algorithm 2 holds.
    else ▷ If the buffer is a spindle source one and *prior* is involved in the spindle data propagation.
      $bf \leftarrow \nabla^m$ ▷ bf is a spindle source buffer.
      $lp \leftarrow LowerP(msgSPT(\mathcal{E}^m))$ ▷ lp is the lower priority among the ones involved in the spindle data propagation.
      $tail \leftarrow$ `getTail`(lp, ∇) ▷ Returns the tail value of lp in bf Equation 5.2
      `readFrom`$(prior, bf, \mathcal{D}, tail)$ ▷ *prior* inherits the tail value from lp in bf.
    end if
  end for
end if
if $\varpi_{prior} = 1$ **then**
  for each $\nabla \in \nabla^{Out}_{prior}$ **do**
    $head \leftarrow$ `getHead`(∇) ▷ **head** has the value of the buffer slot to be written in with respect to a communication resource ∇
    `writeIn`$(prior, \nabla, \mathcal{D}, head)$ ▷ *prior* performs the writing into the buffer slot pointed by **head**
    `setHead`$(prior, \nabla, (1 + head) \mod |\nabla|)$ ▷ Reset to **false** the variable indicating the end of the processing stage.
  end for
  if $prior \equiv lp$ **then** ▷ If *prior* is the lp.
    `setTail`$(prior, bf, (|bf| - 1 + |bf|) \mod |bf|)$ ▷ Tags the new value to be read further from bf Equation 5.1
  end if
end if

Theorem 6 *We consider a message domain* $\mathcal{E}^* = \{\tau_{src}|\tau_1, \cdots, \tau_n, \tau_1^{start}, \cdots, \tau_p^{start}\}$ *such that* ∇^* *is a spindle source buffer where there are* p *tasks involved in data propagation through the spindle and* n *tasks not involved.*

$$SCI = \max(a, b) \tag{5.3}$$

with

$$a = T_{\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}} - \left(c_{\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}} + c_{src}\right) + R_{\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}} \tag{5.4}$$

$$b = \max_{1\leq i\leq n}\{R_i\} \tag{5.5}$$

where c_{src} is the best-case execution time of spindle source task (τ_{src}), $\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$ is the lower priority task among the tasks involved in the spindle data propagation and, accordingly, $T_{\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}}$ and $R_{\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}}$ are its period and its worst-case response time. R_i is the worst-case response time of a task $\tau_i \in \{\tau_1, \cdots, \tau_n\}$.

Proof 7 *Regarding the tasks being part of the spindle, the lifetime of the data samples read from the spindle source buffer goes from the completion of a job of the lower priority task to the completion of the next job. It is intuitively fair to say that the largest time span can only occur if for two instances* p *and* $p+1$*,* p *completes its execution as early as possible and* $p+1$ *as late as possible. One can also say that the largest duration can occur only if at tagging date, this data had just been written and that the instance that produced this data lasted for the shortest possible time. As we consider a pre-emptive system where tasks cannot run in parallel, we combine these two scenarios and find the compromise based on the priority of the producer (*τ_{src}*) and* $\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$.

- *priority(*T_{src}*) > priority(*$\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$*): In these conditions* T_{src} *is likely to preempt* $\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$ *and, logically, in order for* $\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$ *to tag data freshly produced, the execution of* T_{src} *must have occurred during the execution of* $\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$ *given the high priority of* T_{src} *over* $\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$ *and, on the other hand,* $\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$ *must have completed immediately after the completion of* T_{src}*. Therefore, for the execution time*

of $\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$ *to be the shortest possible, the execution of* $\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$ *must have experienced the interference of* τ_{src} *only and the execution of this job of* $\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$ *executes for its best-case execution time. Hence, the largest delay is computed using Equation 5.4.*

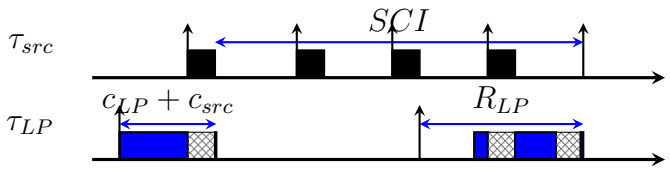

Figure 5.8: *Example displaying* `SCI` *value for a set of tasks*

Finally, in order to deal with all eventualities that could compromise data integrity, it should not be overlooked that the lower priority task among $\{\tau_1, \cdots, \tau_n\}$ *may have a priority lower than that of* $\underset{1\leq c\leq p}{\text{LP}}\left\{\tau_c^{start}\right\}$*. In such a situation, the largest time delay a tagged data sample may remain into the buffer is given by Equation 5.5. Subsequently, given that for the data consistency property, the larger is the amount of time a data sample can remain inside the buffer, the safer it is, taking the maximum of these two values is considered to be a safe solution.*

With respect to the life time bound method, the size of the spindle source buffer is computed on the basis of Theorem 7.

Theorem 7 *We consider* ∇^* *a spindle source buffer written by* τ_{src} *provided the of value of* `SCI`*. The optimal size of* ∇^* *is computed as follows:*

$$|\nabla^*| = \left\lceil \frac{SCI}{T_{src}} \right\rceil \tag{5.6}$$

Proof 8 *The data lifetime bound mechanism necessitates the longest amount of time that a given data sample may be in use by some consumers. Such a time is normally defined by the worst-case response time of a task. Following this logic, if one buffer is shared among several consumers, this duration turns out to be the worst-case response time of the lower priority task among these consumers. Nevertheless, in the case where coherent data matching is required, the need goes beyond the simple tasks response times. In this case, the SCI value*

is the largest amount of time a data may remain in the spindle source buffer of the spindle. Considering that tasks are triggered periodically and that access to the communication buffers is based on the " single producer, several consumers " principle, the size required for such a buffer is clearly the one capable to store all other data samples that may be produced between the completions of the two consecutive jobs of the lower priority task among the ones involved in the spindle data propagation.

In Equation 5.6 we compute the number of instances that can begin and complete their executions in a time bounded by the `SCI` *at the worst. Since each instance gives birth to only one data sample which is written at its execution completion, this confirms that during the DCI, the spindle source task may produce at most n samples provided* $n = \left\lfloor \frac{SCI}{T_{src}} \right\rfloor$. *On the other hand, considering that the instance that marks the propagation of a new sample uses the previously tagged data, which should not be overwritten until the execution completion of its consuming instance, it is mandatory to enlarge the buffer size for one extra slot to maintain the integrity of the data being propagated. For instance, for a buffer comprised of size where the data sample being propagated is located into a slot* $s_{i:i\in[0,size-1]}$, *if n data samples are produced and size is equal to n, the* n^{th} *data sample is written into the slot given by* $(s_i + n)$ mod *size. In other words, the writing of this* n^{th} *data sample overwrites the content of* s_i. *Given that these n samples have been written before the execution the completion of an job of the tagging task that is propagating the sample currently located into the cell* s_i, *if size is equal to n, the overwriting of the data within* s_i *would result in a data consistency property violation. Therefore, another slot must be added.*

5.4.3 Spindle chains propagation delays

In this section we calculate the time it can take for a data sample to propagate from the source until the spindle sink task with respect to each of the chains involved in such a propagation. As indicated previously, this only concerns the chains which are part of the spindle; for the other chains, the data propagation continues to be carried out in an asynchronous way that guarantees the maintenance of data properties required at the level of each pair of adjacent tasks. For the chains involve in the spindle, knowing such delays is useful to determine the data delivery rate for each chain compared to the other chains. These rates allow to

calculate the size of the buffers located at the end of each chain, because this is where the data association should be coherently matched. These time delays are evaluated in two steps:

- **Delay 1** *The time a data sample can be kept into the spindle source buffer provided the additional delay that may be induced by the last reader tags mechanism. For convenience, this time is called* ***`source buffer waiting delay`****.*
- **Delay 2** *The propagation time of a data sample produced by the second task until the completion of the second-to-last task of each spindle chain, considering that the first and last tasks are common for all chains and are the source and the sink tasks of the spindle, respectively. This delay is referred to as* ***`inner propagation delay`****.*

To formalize these delays, we consider the model example presented in Figure 5.4 where we only focus on the chains being part of the spindle. To that end, we start by decomposing the c^{th} spindle chain belonging to the spindle $\mathcal{S}_q^*$ as follows:

$$\mathcal{C}_{c:q}^* = \underbrace{\tau_{src} \rightarrow}_{\text{Delay 1}} \underbrace{\tau_c^{start} \rightarrow \cdots \rightarrow \tau_c^{last} \rightarrow}_{\text{Delay 2}} \tau_{sink} \tag{5.7}$$

where

- τ_{src} and τ_{sink} are the spindle source task and the spindle sink tasks, respectively.
- τ_c^{start} and τ_c^{last} are, respectively, the second and the second-to-last tasks belonging to the c^{th} chain of the spindle, provided $1 \leq c \leq p$.

For each chain $\mathcal{C}_{c:q}^*$, we calculate the minimal and maximal propagation delay for each segment of the chain, that we sum up to obtain the minimal and maximal propagation delay of each chain that we note as $\Omega_{c:q}^{min}$ and $\Omega_{c:q}^{max}$, respectively. By the chain segment we mean the two delays, Delay 1 and Delay 2 associated to the following tasks:

- For the Delay 1, the amount of time that can separate the tagging instant and the consumption of a given data by each task τ_c^{start}. This delay concerns the segment

$$\tau_{src} \rightarrow \tau_c^{start}, \forall c \in [1, p]$$

The minimum and the maximum delays regarding this segment are denoted by swt_c^{min} and swt_c^{max}, respectively.

- Delay 2 concerns the segment being between the activation of τ_c^{start} and the completion of τ_c^{last}

$$\tau_c^{start} \rightarrow \cdots \rightarrow \tau_c^{last}$$

5.4.3.1 The source buffer waiting delay

The smallest amount of time a data sample can spend in the spindle source buffer before being read by a job of τ_c^{start} such that $\tau_c^{start} \neq \underset{1 \leq c \leq p}{\text{LP}} \left\{\tau_c^{start}\right\}$ with $1 \leq c \leq p$ can be neglected and set to zero. A possible scenario would occur if, on one hand, the job of $\underset{1 \leq c \leq p}{\text{LP}} \left\{\tau_c^{start}\right\}$ is preempted by the job of τ_{src}, while the former had only one processor cycle left to run, which is then executed immediately, to be followed immediately by the activation of the job of τ_c^{start}. Accordingly, $\forall 1 \leq c \leq p$

$$swt_c^{min} \approx 0 \tag{5.8}$$

In Figure 5.9 the execution window of the jobs of τ_{src} are colored such that one can see which consumer jobs have used the data sample produced at the execution completion of a corresponding instance of τ_{src}. For example, the execution windows of the jobs of τ_c^{start} and $\underset{1 \leq c \leq p}{\text{LP}} \{\tau_c^{start}\}$ having consumed the data produced at the execution completion of the job of τ_{src} colored in yellow are also of yellow color. At the execution completion of a job of $\underset{1 \leq c \leq p}{\text{LP}} \{\tau_c^{start}\}$ a new data sample is tagged and all the other jobs activated after this tagging instant start consuming the new tagged data. Moreover, the two rods represent scenarios whereby the executions of the three jobs have the smallest variation between them which can be approximated to zero. The red rod shows the case in which this small deviation has separated the execution completion of the first job of τ_{src}, which is followed by the resume and the completion of the first job of $\underset{1 \leq c \leq p}{\text{LP}} \{\tau_c^{start}\}$, followed in its turn by the activation of the first job of τ_c^{start}. Thus, the just released job of τ_c^{start} consume a data sample freshly written by this job of τ_{src} and tagged at the execution completion of this instance of $\underset{1 \leq c \leq p}{\text{LP}} \{\tau_c^{start}\}$. Accordingly, for such a scenario, the time variation between these three operations is closer to one computed by Equation 5.8.

On the other hand, each τ_c^{start} such that $\tau_c^{start} \neq \underset{1 \leq c \leq p}{\text{LP}} \left\{\tau_c^{start}\right\}$ with $1 \leq c \leq p$ has a higher priority over $\underset{1 \leq c \leq p}{\text{LP}} \left\{\tau_c^{start}\right\}$. That being, no job of $\underset{1 \leq c \leq p}{\text{LP}} \left\{\tau_c^{start}\right\}$ can execute if there is one of

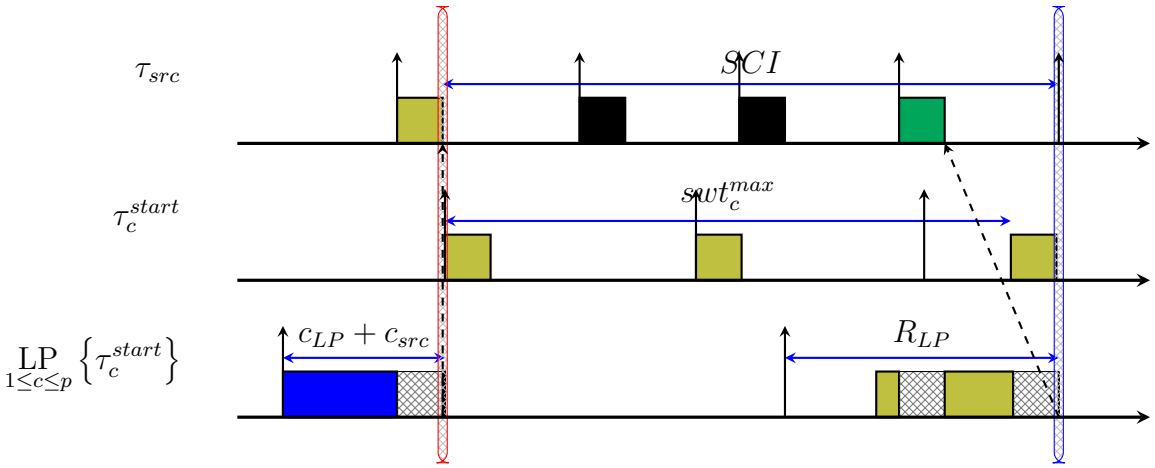

Figure 5.9: *Example displaying the swt_c^{min} and swt_c^{max} values for each τ_c^{start}*

τ_c^{start} still executing. On the other hand, a data sample to be propagated by the job(s) of any τ_c^{start} must be kept in use until the completion of the current job of $\underset{1\leq c\leq p}{\mathrm{LP}}\left\{\tau_c^{start}\right\}$. Considering that each job must read all the input data at the activation time, the largest amount of time a data sample can remain into the spindle source buffer is closer to $SCI - c_c^{start}$ when the last job of τ_c^{start} hasn't encountered any interference from higher priority tasks and its execution lasts for the best-case time value. Hence, $\forall 1 \leq c \leq p$

$$swt_c^{max} \approx SCI - c_c^{start} \tag{5.9}$$

where c_c^{start} is the best-case execution time of τ_c^{start}.

The vertical blue rod shows the scenario in which the job of τ_c^{start} preempts a job of $\underset{1\leq c\leq p}{\mathrm{LP}}\left\{\tau_c^{start}\right\}$ such that at the end of its execution, the preempted job resumes, instantly completes and tags the new data to be propagated. Therefore, the preempting job is the last one of those having propagated the previously tagged data. If we consider that the reading of the data is performed at the job activation time, if this preempting job executed for its best-case time value with no interference from jobs of the higher priority tasks, then the largest delay that this job consumed will have been into spindle source buffer for a time value given by Equation 5.9.

5.4.3.2 The inner propagation delay

The propagation delay that we calculate here concerns the segment $\tau_c^{start} \rightarrow \cdots \rightarrow \tau_c^{start}$. This delay consists of the time that can elapse from the moment a job of τ_c^{start} starts its execution until a data related to the one read from the spindle source buffer is written by a job of any τ_c^{last}.

Regarding the minimal delay, we start from the activation of the first job of τ_c^{start} having read a given data sample the earliest possible (Equation 5.8), whereas for the maximum delay we start with the activation of the last job of τ_c^{start} having propagated the same data (Equation 5.9). Thus, for each pair of adjacent tasks τ_i and τ_j with $(\tau_i, \tau_j) \in \{\tau_c^{start} \rightarrow \cdots \rightarrow \tau_c^{start}\}$ such that $\tau_i \rightarrow \tau_j$, the smallest propagation delay of a data sample from the activation of a job of τ_i until the completion of a job of τ_j, denoted by $Delay_{i,j}^{min}$, occurs if the jobs of τ_i and τ_i are activated as soon as possible, run for their shortest time (best-case execution time) such that the completion of the job of τ_i corresponds to the activation of the job of τ_i. Formally,

$$Delay_{i,j}^{min} = c_i + c_j \tag{5.10}$$

where c_i and c_j are, respectively, the best-case execution time of τ_i and τ_j.

Subsequently, for all $\{\tau_i, \tau_{i+1}, \cdots, \tau_n\} \equiv \left\{\tau_c^{start}, \cdots, \tau_c^{last}\right\}$ such that $\tau_i \rightarrow \tau_{i+1}$, the smallest delay being between the activation of a job of τ_i and

$$Delay_{i,n}^{min} = \sum_{1 \leq i \leq n} c_i \tag{5.11}$$

Likewise, we need to calculate the largest amount of time that can elapse between the activation instant of a job of τ_c^{start} and the execution completion of a job of τ_c^{last} which ends with the writing of a new data sample (related to the message read from the source buffer) into the corresponding spindle chain last buffer. At this stage we already know the largest amount of time that a tagged data sample can pass before a job of of τ_c^{start} (or its last job willing to read it) may be able to propagate it. We denoted this value as swt_c^{max} which we computed using Equation 5.9.

In the following, we analogically denote by $Delay_{i,n}^{max}$ the largest propagation delay regarding the segment being between τ_c^{start} and τ_c^{last} with respect to the c^{th} spindle propagation chain. To do so, we first respond to the following: *"what is the largest amount of time a data can remain available into the buffer before a new data sample is written"* and, *"when should the consumer read this data for its propagation to be the largest possible"*. Responding to the first question, Becker and al. in [31] studied the job-level data dependencies for automotive multi-rate effect-chains. Therein, for each two adjacent tasks τ_i and τ_j such that τ_i produces input for τ_j, the maximum delay a data sample produced by a job of τ_i can be available for the job of τ_j is utmost $2T_i - C_i$ where C_i and T_i are the worst-case execution time and the period of τ_i respectively. This delay becomes $2T_i - c_i$ with c_i if the execution lasts for the best-case execution time. This corresponds to the scenario where, for two consecutive jobs of τ_i, the first completes its execution as early as possible whereas the second completes as late as possible. Thus, in order for the propagation of the data sample produced by a job of τ_i to last for as long as possible, the job τ_j (if only one job can propagate that data) or the last job of τ_j (if several jobs of τ_j can propagate a data sample produced by τ_i, at worst) must be triggered after the data produced by the job of τ_i has spent $2T_i - c_i$ time units into the corresponding buffer and executes such that the data it will produce remain available for the corresponding consumers for as log as possible.

To summarize all this, no matter whether a data is used by one or more jobs of the consumer, the eligibility period for a data to be consumed depends on the time separating two consecutive outputs of the producer. It is also obvious that the upper bound time between the earliest activation of one job and the output of data by the next job (which generates a new data sample), related to a some task, is 2 times the period. Accordingly, for all $\{\tau_i, \tau_{i+1}, \cdots, \tau_n\}$ such that $\tau_i \rightarrow \tau_{i+1}$, the largest delay being between the activation of a job of τ_i and the one of τ_n is computed as follows:

$$Delay_{i,n}^{max} = \sum_{1 \leq i \leq n-1} 2 * T_i \tag{5.12}$$

To conclude, for the c^{th} spindle propagation chain, the smallest and largest time that may separate the reading of a data sample from the spindle source buffer and the writing of a

corresponding data sample into the spindle chain last buffer are, respectively, calculated as follows

$$\Omega_{c:q}^{min} = swt_c^{min} + Delay_{i,n}^{min} \tag{5.13}$$

and

$$\Omega_{c:q}^{max} = swt_c^{max} + Delay_{i,n}^{max} \tag{5.14}$$

5.4.4 The scroll or overwrite mechanism

The scroll or overwrite mechanism (`SorO`) applies only to the second-to-last task of each spindle chain provided that such tasks are the ones in charge of writing into the corresponding spindle chain last buffers from where the jobs of the spindle sink task must retrieve matching data samples. Depending on the propagation delay of each spindle chain, it is highly likely that several data samples resulting from the same execution step of τ_{src} may be in the spindle chain last buffers, thereby increasing the required space unnecessarily. The goal of this mechanism consists in providing each spindle chain second-to-last task with the capability to decide either to write into the next slot of the buffer or to overwrite the content of the slot in which the previous writing took place. Hence the name `"scroll or overwrite"`. In other words, if the previously written data and the data ready to be written all initially result from the same execution step of τ_{src} the previously written data sample by a job of a given τ_c^{last} is replaced by the new data (`overwrite`[2]), otherwise, write this new data sample in the next buffer slot (`scroll`) in the sense of `"move to the following slot"`.

Intuitively, all the data samples resulting from the same execution step of τ_{src} will occupy the same buffer slot by continuously overwriting each other until another data sample resulting from another execution step (of τ_{src}) is generated.

5.4.4.1 Data sample tracking

The implementation of the `SorO` mechanism takes advantage on the data double timestamping and the implementation scheme of the circular buffer as described in the Sections 3.2.3 and

[2]It may be decided to discard the writing while keeping the previous written data

3.3.1, respectively. The data double timestamping scheme is depicted in Figure 3.8.

Speaking of the data double timestamp, within a spindle, it is considered that a timestamp value is added to each data sample produced by the spindle source task and all the data samples, resulting from it, own and keep this timestamp value unchanged until the spindle sink task. While keeping the timestamp value constant, each task, involved in the propagation of a timestamped data by the source task, adds to its output data a `timeOfIssue` which normally corresponds to the job execution completion time. So, since data samples reaching the spindle sink task will have a common faraway origin, the `SorO` mechanism acts in the sense of giving the spindle sink task the possibility to match them on the basis of their timestamps.

5.4.4.2 Computing the spindle chain last buffers size

According to this mechanism all data samples with the same timestamp will occupy alternately the same buffer slot. Each output data at the end of each second-to-last task of each chain will be available no earlier than a traversal time $\Omega_{c:q}^{min}$ and no later than a traversal time $\Omega_{c:q}^{max}$. Sometimes, while the chains with the longer propagation delay are still working on producing the first output, the ones with shorter propagation delays may have produced several outputs. Such output data, produced in the meantime by the chains with shorter propagation delays, may originate from the same execution step of the spindle source task or from different execution steps of the latter. In case they originate from the same execution step of the spindle source task, they must have occupied the same buffer slot. At worst, if they all came from different execution steps of the spindle source task, they would have occupied the same number of different slots as there was output in the meantime. The size required for each buffer at the end of each chain should be the one allowing to handle the worst case: `when all the output data are coming from different execution steps of the spindle source task`.

To guarantee that there will always be matching data, it is necessary to have cells where each second-to-last task of each chain is going to write its new output data until the propagation through the slowest path reaches the end of the corresponding chain. To meet this requirement, we formulate Theorem 8.

Theorem 8 *We consider the spindle source τ_{src}, the spindle sink τ_{sink} and the tasks $\tau_1^{last}, \cdots, \tau_q^{last}$ where q is the number of spindle chains propagating the data samples related to a given message from τ_{src} until τ_{sink}. For each spindle chain $\mathcal{C}_{c:q}^*$, the optimal size of the corresponding spindle chain last buffer denoted by $|\nabla^{last_c^*}|$ is given by*

$$|\nabla^{last_c^*}| = \max\left(\left\lceil \frac{\max\limits_{1\leq c\leq q}\Omega_{c:q}^{max}}{\Omega_{c:q}^{max}} \right\rceil, \left\lceil \frac{R_{sink}}{T_c^{last}} \right\rceil\right) \tag{5.15}$$

where

- $\max\limits_{1\leq c\leq q}\Omega_{c:q}^{max}$ *is the largest delay it can take to have an output data sample resulting from a data sample read from the spindle source data, all the q spindle chains considered together.*
- $\Omega_{c:q}^{min}$ *is the shortest delay it can take to have an output data sample resulting from a data sample read from the spindle source data with respect to the c^{th} chain out of q chains.*
- T_c^{last} *is the period of the second-to-last task belonging to the c^{th} chain out of q chains.*
- R_{sink} *is the worst-case response time of the spindle sink task.*

Proof 9 *Each chain $\mathcal{C}_{c:q}^*$ is compared with the chain with the largest propagation delay. The latter is the one that is likely to be late while one or more output data may already have been produced at the output of $\mathcal{C}_{c:q}^*$. On the other hand, since all data from the same execution step of the spindle source task will occupy the same buffer slot, at worst, $\left\lceil \frac{\max\limits_{1\leq c\leq q}\Omega_{c:q}^{max}}{\Omega_{c:q}^{max}} \right\rceil$ output data resulting from different execution steps of the spindle source task are likely to have been written at the end of $\mathcal{C}_{c:q}^*$ when the slowest propagation of a single output data arrives at the end of the its chain. Accordingly, the ratio between $\max\limits_{1\leq c\leq q}\Omega_{c:q}^{max}$ and $\Omega_{c:q}^{max}$ represents the size required for the spindle chain last buffer located at the end of the c^{th} chain so as to ensure that there will always be matching data in each input buffer for the sink task. To achieve this, the output data from the fastest chains, resulting from other execution steps of the spindle source task, must have a place to be buffered until there are such matching data in all input buffers for the spindle sink task. On the other hand, it may happen that $\left\lceil \frac{\max\limits_{1\leq c\leq q}\Omega_{c:q}^{max}}{\Omega_{c:q}^{min}} \right\rceil < \left\lceil \frac{R_{sink}}{T_c^{last}} \right\rceil$. Therefore, considering that the access to the data is done asynchronously without blocking on*

the shared resource, it is mandatory to consider the maximum of these two ratios in order to maintain the data consistency property as well. Implementation of this mechanism is described by the Algorithm 4

Algorithm 4 *Implementation of the scroll or overwrite mechanism*

```
1: function PREVIOUSTSTAMP(Buffer buf)
2:     return previousTimestamp                          ▷ Returns previous timestamp from buf.
3: end function
4: return the higher priority task prior and its output buffers ∇^Out_prior
5: if ϖ_prior = 1 then          ▷ We assume prior is a second-to-last task and it enters the writing stage.
6:     int head
7:     for each ∇ ∈ ∇^Out_prior do
8:         head ← getHead(∇)   ▷ head points to the slot where to write normally (when the SorO mech. is not considered).
9:         if D->timestamp = previousTStamp(∇) then      ▷ Do the previous and the current data have same timestamp?
10:            head ← (head − 1 + |∇|) mod |∇|            ▷ If true, don't step forward, write where wrote previously.
11:        end if                                         ▷ If false, consider the current value of head (Statement 8).
12:        writeIn(prior, ∇, D, head)                     ▷ prior performs the writing into the buffer slot pointed by head
13:        setHead(prior, ∇, (1 + head) mod |∇|)          ▷ Reset temporally the new value of head where the next job will write.
14:    end for
15: end if
```

Example 3 *An application example is depicted on Figure 5.10 where all tasks have implicit deadlines. For simplicity we consider that jobs are only characterized by the worst-case execution times and the period. So wherever the best-case execution time is required, we'll substitute it by the worst-case execution time.*

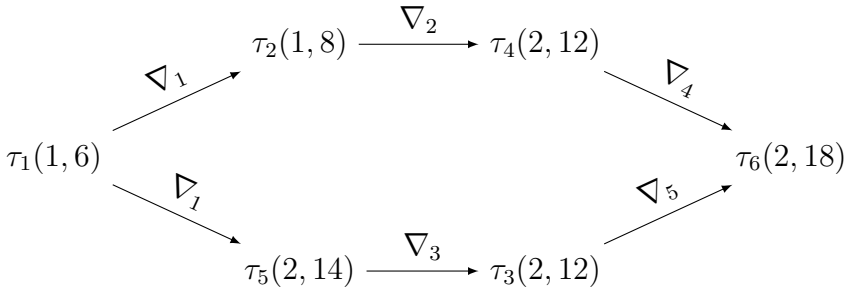

Figure 5.10: *A balanced spindle application example.*

Given that $priority(\tau_5) < priority(\tau_2)$*, to compute the following parameters the following way: Using the ,*

$$\begin{cases} SCI = T_5 - (C_5 + C_1) + R_5 = 14 - (2+1) + 10 = 21 & , Equation\ 5.3 \\ |\nabla_1| = \left\lfloor \frac{SCI}{T_{src}} \right\rfloor + 1 = \left\lfloor \frac{21}{6} \right\rfloor + 1 = 4 & , Equation\ 5.6 \\ |\nabla_2| = \left\lceil \frac{R_4}{T_2} \right\rceil = \left\lceil \frac{6}{8} \right\rceil = 1 & , Equation\ 4.1 \\ |\nabla_3| = \left\lceil \frac{R_3}{T_5} \right\rceil = \left\lceil \frac{4}{14} \right\rceil = 1 & , Equation\ 4.1 \end{cases}$$

From the spindle model presented in Figure 5.10, we retrieve the following spindle chains:

$$\begin{cases} \mathcal{C}^*_{1:2} = \tau_1 \xrightarrow{\nabla_1} \tau_2 \xrightarrow{\nabla_2} \tau_4 \xrightarrow{\nabla_4} \tau_6 \\ \mathcal{C}^*_{2:2} = \tau_1 \xrightarrow{\nabla_1} \tau_5 \xrightarrow{\nabla_3} \tau_3 \xrightarrow{\nabla_5} \tau_6 \end{cases}$$

So, regarding the $\mathcal{C}^*_{1:2}$ we have the following results

$$\begin{cases} swt_1^{min} = 0 & , Equation\ 5.8 \\ swt_1^{max} = SCI - C_2 = 21 - 1 = 20 & , Equation\ 5.9 \\ Delay_{2,4}^{min} = C_2 + C_4 = 1 + 2 = 3 & , Equation\ 5.11 \\ Delay_{2,4}^{max} = 2 * T_2 + 2 * T_4 = 16 + 24 = 40 & , Equation\ 5.12 \\ \Omega_{1:2}^{min} = swt_1^{min} + Delay_{2,4}^{min} = 3 & , Equation\ 5.13 \\ \Omega_{1:2}^{max} = swt_1^{max} + Delay_{2,4}^{max} = 20 + 40 = 60 & , Equation\ 5.14 \end{cases}$$

while for $\mathcal{C}^*_{2:2}$ we have

$$\begin{cases} swt_2^{max} = 0 & , Equation\ 5.8 \\ swt_2^{max} = SCI - C_5 = 21 - 2 = 19 & , Equation\ 5.9 \\ Delay_{3,5}^{min} = C_3 + C_5 = 2 + 2 = 4 & , Equation\ 5.11 \\ Delay_{3,5}^{max} = 2 * T_3 + 2 * T_5 = 24 + 28 = 52 & , Equation\ 5.12 \\ \Omega_{2:2}^{min} = swt_2^{min} + Delay_{3,5}^{min} = 4 & , Equation\ 5.13 \\ \Omega_{2:2}^{max} = swt_2^{max} + Delay_{3,5}^{max} = 19 + 52 = 71 & , Equation\ 5.14 \end{cases}$$

We now compute the size of each of the spindle chain last buffer(∇_4 and ∇_5) using Equation 5.15 as follows

$$\begin{cases} |\nabla_4| = \max\left(\lceil \frac{\max\left(\Omega_{1:2}^{max}, \Omega_{2:2}^{max}\right)}{\Omega_{1:2}^{max}} \rceil, \lceil \frac{R_6}{T_4} \rceil\right) = \max\left(\lceil \frac{71}{60} \rceil, \lceil \frac{12}{12} \rceil\right) = 2\ slots \\ |\nabla_5| = \max\left(\lceil \frac{\max\left(\Omega_{1:2}^{max}, \Omega_{2:2}^{max}\right)}{\Omega_{2:2}^{max}} \rceil, \lceil \frac{R_6}{T_3} \rceil\right) = \max\left(\lceil \frac{71}{71} \rceil, \lceil \frac{12}{12} \rceil\right) = 1\ slots \end{cases} \quad (5.16)$$

From Equation 5.16 we know that the buffer being at the end of the spindle chain with a largest propagation delay will always equal to one. We close this chapter by proposing the Algorithm 5 allowing the spindle sink to associate only matching data.

As an illustrating example, we consider the scheduling results presented in Figure 5.11. Here we can see the propagation path of each tagged data from the spindle source buffer until the corresponding spindle chain last buffer. The vertical rods indicate the execution completion of the job of τ_5 that tagged the data sample previously written by the job of τ_1. Depending on the gaps between periods of the tasks constituting the inner segment of each propagation chain, some data samples, even though they have been initially tagged, may be overwritten before reaching the spindle sink task. Therefore, what is important for us is that, for those who have reached the end of the chain, the sink task must be able to find there those resulting from the same execution step of the source task. For instance, the propagation of the data sample marked in red managed to reach the end of the chain $\mathcal{C}^*_{2:2} = \tau_1 \xrightarrow{\nabla_1} \tau_5 \xrightarrow{\nabla_3} \tau_3 \xrightarrow{\nabla_5} \tau_6$ while the propagation through the chain $\mathcal{C}^*_{1:2} = \tau_1 \xrightarrow{\nabla_1} \tau_2 \xrightarrow{\nabla_2} \tau_4 \xrightarrow{\nabla_4} \tau_6$ is suspended by the fourth job of τ_2, activated after the completion of the second job of τ_5 (in red) which tagged the green colored data sample.

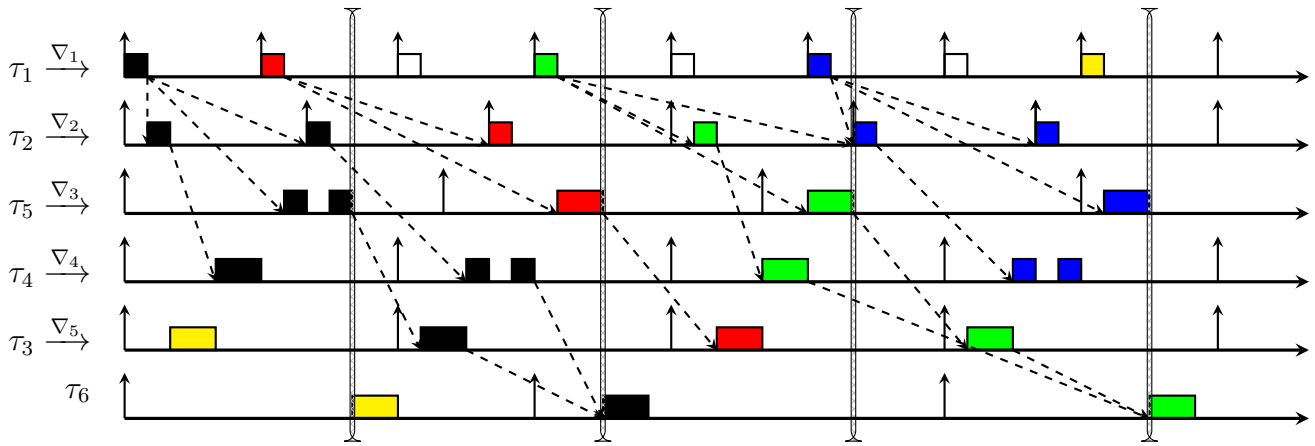

Figure 5.11: *The global consistency verification process example*

As a result, at the activation time of the third job of τ_6, it must read the data samples resulting from the same execution step of τ_1 still in buffers ∇_4 and ∇_5. Therefore, only the green matches are available in both ∇_4 and ∇_5. In all the cases shown in this figure, at each activation of a job of τ_6 there were always matching data in each of the buffers (∇_4 and ∇_5)

and between two successive activation of its jobs, the data samples having been written into ∇_4 and ∇_5 never exceed a maximum of two data samples into each.

Algorithm 5 *Data matching algorithm for the spindle sink task*

1: **function** PREVIOUSTSTAMP(Buffer buf)
2: **return** previousTimestamp ▷ Returns previous timestamp from buf.
3: **end function**
4: **function** SPLASTBUFFERS(setOfinputBuffers) ▷ Retrieves the set of spindle chain last buffers from a set of buffers.
5: **return** SCLB ▷ SCLB= the set of Spindle Chain Last Buffers.
6: **end function**

7: **return** the higher priority task *prior* and its input buffers ∇^{In}_{prior}
8: **if** $\varpi_{prior} = -1$ **then** ▷ We assume that *prior* is the spindle sink task and it enters the reading stage.
9: **Buffer** $SpLCB[\,] \leftarrow SPLastBuffers(\nabla^{In}_{prior})$
10: **Buffer** $min\nabla \leftarrow min\,(SpLCB)$ ▷ SpLCB contains a list of spindle chain last buffers retrieved from the set ∇^{In}_{prior}.
11: **int** $timestamp \leftarrow previousTStamp(min\nabla)$ ▷ The timestamp of the recently inserted data into $min\nabla$.
12: **int** *tail* ▷ The variable **tail** holds the number of the slot from where to read.
13: **for** each $\nabla^m \in \nabla^{In}_{prior}$ such that $1 \leq m \leq |\nabla^{In}_{prior}|$ **do**
14: **if** $\nabla^m \notin SpLCB$ **then**
15: Follow Algorithm 2 ▷ For buffers not involved in the spindle data propagation, refer to the Algorithm 2.
16: **else** ▷ Otherwise, fetch and associate only matching data.
17: **if** `readEnd` $= False$ **then** ▷ If prior is just activated (is at the beginning of its execution), fetch matching data.
18: **for** each $\nabla \in SpLCB$ **do**
19: **if** $\mathcal{D}$->$timestamp = timestamp$ **then** ▷ Check the slot containing the sample with this timestamp.
20: $tail \leftarrow getTail(prior, \nabla)$ ▷ Return the number of such a buffer slot.
21: $readFrom(prior, \nabla, \mathcal{D}, tail)$ ▷ Read from this position with respect to ∇.
22: `setTail`$(prior, \nabla, (1 + tail) \mod |\nabla|)$ ▷ Set this value to be pointed to during this execution window.
23: **end if**
24: **end for**
25: **else** ▷ Otherwise, read from the slot pointed by tail value previously set at Statement 22).
26: `readFrom`$(prior, \nabla^m, \mathcal{D}^m, tail)$
27: **end if**
28: **end if**
29: **end for**
30: `readEnd` $\leftarrow True$ ▷ Mention the fact that reading has taken place for at-least one time.
31: **end if**

Chapter

6 Conclusions and perspectives

Real-time embedded systems, despite their limited resources, are evolving very quickly. This evolution is characterized by the consideration of increasingly intelligent functionalities aimed at rendering their functioning completely autonomous. This applies, for instance, to autonomous vehicles, drones, robots, etc. This functioning autonomy in terms of decision making depends on the quality of the collected data regarding the surrounding environment state. Therefore, for such systems, it is not enough to ensure that all jobs do not miss their deadlines, it is also mandatory to ensure the good quality of the data being transmitted from tasks to tasks. Speaking of the data quality constraints, they are expressed by the maintenance of a set of properties that a data sample must exhibit to be considered as relevant. To ensure the overall performance of these systems, designers must therefore solve the issue of finding trade-offs between the system scheduling constraints and those applied to the data. As the scheduler does not take into account data constraints, the operating system introduces mechanisms aiming to arbitrate the access communication resources (shared resources). The potential effects of introducing these mechanisms include the formation of deadlocks and significant blocking times which, in turn, may negatively impact the system scheduling. After investigating the mechanisms widely proposed in the literature, in this thesis, we opted for the `wait-free mechanism` for their ability not to induce blockages between tasks accessing the same communication resource. The size of each communication buffer is based on the `the lifetime bound method` which relies on the timing parameters of the producer and the consumers for a given type of message. The access to the communication buffer dedicated to each type of message is done following the `single writer, many readers`. Afterwards, we analyzed the specificities of the μORB communication system (implemented in the PX4 and PX4-RT) to propose the adaptations that are necessary to meet the data constraints

considered in this thesis. Major specificities include the fact that a same task can produce messages of different types and that, among these types of messages, the producer task can send each type of message to a different list of consumer tasks (e.g. the message of type m_1 can be sent to tasks τ_1 and τ_2, type m_2 to τ_1 and τ_4, type m_3 to τ_4 and τ_5, etc). Besides, it is possible to find that a message of a given type can be produced by different tasks (multi-source message). Due to these specificities and so as to bound the size of the buffers dedicated to each type of message, we modeled the interactions between the tasks by a bipartite graph that we called `communication graph`. Using this graph, for each type of message, the set consisting of its producer and consumers is clearly identifiable. Such a set of tasks has been named `message domain` and formalized. Still on the modeling dimension (task system and communication system modeling), to enhance the predictability of inter-task communication, we extend Liu and Layland model with the parameter `communication state`. The benefit of this parameter lies in the fact that it allows to efficiently control at which execution points the write/read operations are performed. Among the future perspectives, with the help of this parameter, we plan to formally evaluate the communication cost of the communication mechanisms proposed in this thesis.

The second part deals with the verification of the data constraints. We considered the maintenance of data properties at the local level (between adjacent tasks) and at the global level (considering the propagation of data throughout the functional chain). Consequently, we set two constraints, namely, `data local constraints` and `data global constraints`. The local constraint requires that the data exchanged between adjacent tasks must be `fresh` (recently produced at the activation time of a job of a consuming task), `locally matching` (for the category of `multi-source` messages, the consumer must take the `fresh data` from each of the buffers written by these producers) and `consistent` (simply, a data sample in use by at least one of the jobs of the consumer tasks must never be overwritten). Regarding the `data global constraint`, it refers to the parts of the system (sub-systems) where a data produced by a single task is propagated across various functional chains to converge on a same task which associates the output data from each chain. The propagation paths (chains) of these data form what we call a `spindle` and the chains involved in this propagation are called `spindle chains`. The first task in the spindle is called the `spindle source`

`task` while the task at the end of the spindle has been called the `spindle sink task`. The global constraint is verified only if the sink task must associate the data originally resulting from the same execution step of the spindle source task. To verify the `data local constraints` we introduced a mechanism referred to as `sub-sampling mechanism` which maintains the three data properties required at the local level (freshness, local matching and consistency). The novelty of this mechanism is that, at the same instant, different tasks have the possibility to read data from different slots in the buffer. In a multi-rate system the tasks can have different periods. Therefore, this mechanism allows the tasks with smaller periods to read newly produced data while the slower tasks may still be using other data (with no arbitration mechanism required). Regarding the verification of `data global constraint`, two other mechanisms have been introduced, `the last reader tags mechanism` (defining which data produced by the spindle source task must be propagated) and the `scroll or overwrite mechanism` (providing the second-to-last task on each spindle chain the data writing guidelines). These two mechanisms are to some extent complementary. The first one works at the beginning of the spindle while the second one works at the end of the spindle.

Considering that the propagation of some data produced by the spindle source task can be disrupted before reaching the spindle sink task, `the last reader tags mechanism` filters and allows the propagation of those having a high chance to reach the sink task. Furthermore, thanks to the `data double timestamping`, all the data originating from a same data sample produced by the spindle source task have a same timestamp value. Exploiting this property, the `scroll or overwrite mechanism` provides the second-to-last task of each spindle chain with the ability to decide if the new output data has to overwrite the recently written one (if they have the same timestamp value) or has to be written in the next slot of the buffer (otherwise). The key advantage of this approach lies in the minimisation of the size of the buffers being at the end of each spindle chain by keeping only a single copy of data among those resulting from the same execution step of the spindle source task. The sizes of the buffers being at the end of each spindle chain are calculated so that, at the activation time of a job of the spindle sink task, there are always `matching data` in each of them. An algorithm allowing the association of `matching data` is proposed.

In the future we expect to extend the global constraint to α-`balanced spindle` where

there are nested spindles within others. Also, the results will be extended to the association of data from different sources. Last but not least, in the near future, we envisage to implement the contributions of this thesis on the PX4-RT autopilot.

Bibliography

Chapter 1: The real-time domain

[1] J. Stankovic S. Biyabani and K. Ramamrithan. "The integration of deadline and criticalness in hard real-time scheduling". In: *In preceedings of the IEEE Real-Time Systems Symposium*. 1988 (Cited on page 5).

[2] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* (1973) (Cited on pages 5, 9, 16, 60, 61, 64, 103).

[3] Reinhard Wilhelm et al. "The worst-case execution-time problem, overview of methods and survey of tools". In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008), p. 36 (Cited on page 6).

[4] Robert I. Davis and Liliana Cucu-Grosjean. "A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems". In: *LITES* (2019) (Cited on page 6).

[5] M. Joseph and P. Pandya. "Finding Response Times in a Real-Time System". In: *Comput. J.* (1986) (Cited on pages 6, 14, 16, 77).

[6] Robert I. Davis and Alan Burns. "A survey of hard real-time scheduling for multiprocessor systems". In: *ACM Comput. Surv.* (2011) (Cited on page 7).

[7] AUTOSAR. "Spec. of Timing Extensions". In: *AUTOSAR Std. 4.3* (2016) (Cited on pages 8, 9, 27, 28, 55).

[8] Joël Goossens, Emmanuel Grolleau, and Liliana Cucu-Grosjean. "Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms". In: *Real-Time Systems* 52.6 (2016), pp. 808–832 (Cited on page 8).

[9] John Carpenter et al. “A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms”. In: *Handbook of Scheduling - Algorithms, Models, and Performance Analysis*. 2004 (Cited on page 8).

[10] Joseph Y.-T. Leung and Jennifer Whitehead. “On the complexity of fixed-priority scheduling of periodic, real-time tasks”. In: *Perform. Evaluation* (1982) (Cited on pages 9, 60).

[11] OSEK. *OSEK/VDX Operating Systems . ver. 2.2.3, OSEK, 2006*. http://www.osek-vdx.org (Cited on page 9).

[12] P. Brinch Hansen. “Operating System Principles”. In: *Prentice-Hall*. 1973 (Cited on page 13).

[13] J. Peterson and A. Silberschatz. “Operating Systems Concepts.” In: *Addison-Wesley*. 1985 (Cited on page 13).

[14] Paolo Torroni, Zeynep Kiziltan, and Eugenio Faldella. “Blocking time under basic priority inheritance: Polynomial bound and exact computation”. In: *CoRR* abs/1806.01589 (2018) (Cited on pages 15, 16).

[15] S. Baranov and V. Nikiforov. “The impact of blocking factor on real-time applications feasibility”. In: *2016 18th Conference of Open Innovations Association and Seminar on Information Security and Protection of Information Technology (FRUCT-ISPIT)*. 2016 (Cited on pages 15, 16).

[16] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”. In: *IEEE Trans. Computers* (1990) (Cited on pages 15, 16, 21).

[17] Tomasz Kloda, Antoine Bertout, and Yves Sorel. “Latency analysis for data chains of real-time periodic tasks”. In: *23rd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2018, Torino, Italy, September 4-7, 2018*. 2018, pp. 360–367 (Cited on pages 15, 16).

[18] Johannes Schlatow and Rolf Ernst. “Response-Time Analysis for Task Chains in Communicating Threads”. In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*. 2016, pp. 245–254 (Cited on pages 15, 16).

[19] Y. Ding J. Lehoczky L. Sha. “The rate monotonic scheduling algorithm: Extact characterization and average case behavior”. In: *Real-Time Systems Symposium* (1989) (Cited on page 16).

[20] Francisco J. Cazorla et al. “PROARTIS: Probabilistically Analyzable Real-Time Systems”. In: *ACM Trans. Embed. Comput. Syst.* (2013) (Cited on page 17).

[21] Yue Lu et al. “A new way about using statistical analysis of worst-case execution times”. In: *SIGBED Rev.* (2011) (Cited on page 17).

[22] Sebastian Altmeyer, Liliana Cucu-Grosjean, and Robert I. Davis. “Static probabilistic timing analysis for real-time systems using random replacement caches”. In: *Real Time Syst.* (2015) (Cited on page 17).

[23] Sebastian Altmeyer and Robert I. Davis. “On the correctness, optimality and precision of Static Probabilistic Timing Analysis”. In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*. Ed. by Gerhard P. Fettweis and Wolfgang Nebel (Cited on page 17).

[24] Julien Forget, Frédéric Boniol, and Claire Pagetti. “Verifying end-to-end real-time constraints on multi-periodic models”. In: *22nd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2017, Limassol, Cyprus, September 12-15, 2017*. 2017, pp. 1–8 (Cited on pages 17, 18, 23).

[25] Michaël Lauer et al. “Latency and freshness analysis on IMA systems”. In: *IEEE 16th Conference on Emerging Technologies & Factory Automation, ETFA 2011, Toulouse, France, September 5-9, 2011*. 2011 (Cited on pages 17, 29, 31).

[26] Michaël Lauer et al. “Worst Case Temporal Consistency in Integrated Modular Avionics Systems”. In: *13th IEEE International Symposium on High-Assurance Systems En-*

gineering, HASE 2011, Boca Raton, FL, USA, November 10-12, 2011. 2011 (Cited on pages 17, 30).

[27] Michaël Lauer. "Une méthode globale pour la vérification d'exigences temps réel : application à l'Avionique Modulaire Intégrée. (A comprehensive method for real-time requirements verification : application to Integrated Modular Avnionics)". PhD thesis. National Polytechnic Institute of Toulouse, France, 2012 (Cited on pages 17, 29).

[28] Rémy Wyss et al. "End-to-end latency computation in a multi-periodic design". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*. 2013 (Cited on page 17).

[29] Arne Hamann et al. "Communication Centric Design in Complex Automotive Embedded Systems". In: *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*. LIPIcs. 2017 (Cited on pages 17, 19, 28).

[30] Alix Munier Kordon and Ning Tang. "Evaluation of the Age Latency of a Real-Time Communicating System Using the LET Paradigm." In: *In preceedings of the ECRTS*. 2020 (Cited on pages 17, 28).

[31] S Mubeena M Becker D Dasari, M Behnam, and Thomas Nolte. "Synthesizing Job-Level Dependencies for Automotive Multi-Rate Effect Chains." In: *RTCSA, At Daegu, South Korea*. 2016 (Cited on pages 17, 114).

[32] S Mubeena M Becker D Dasari, M Behnam, and Thomas Nolte. "End-to-end timing analysis of cause-effect chains in automotive embedded systems." In: *Journal of Systems Architecture*. 2017 (Cited on page 17).

[33] Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. "Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study". In: *Comput. Sci. Inf. Syst.* 10.1 (2013), pp. 453–482 (Cited on page 17).

[34] Guohui Li et al. "Maintaining Data Freshness in Distributed Cyber-Physical Systems". In: *IEEE Trans. Computers* (2019) (Cited on page 18).

[35] Ricardo Santo Marques, Nicolas Navet, and Françoise Simonot-Lion. "Configuration of in-vehicle embedded systems under real-time constraints. In Proceedings of the 10*th* IEEE International Conference on Emerging Technologies and Factory Automation, ETFA'05, Catania, Italy. IEEE, 2005." In: 2011 (Cited on page 18).

[36] N. Feiertag et al. "A Compositional Frameworfor End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics". In: *CRTS* (2008) (Cited on pages 18, 28).

[37] Krithi Ramamritham, Sang Hyuk Son, and Lisa Cingiser DiPippo. "Real-Time Databases and Data Services". In: (2004) (Cited on page 18).

[38] Dimitri Theodoratos and Mokrane Bouzeghoub. "Data Currency Quality Satisfaction in the Design of a Data Warehouse". In: (2001) (Cited on page 18).

[39] Mokrane Bouzeghoub. "A Framework for Analysis of Data Freshness". In: Paris, France: Association for Computing Machinery, 2004 (Cited on page 19).

[40] Mokrane Bouzeghoub and Verónika Peralta. "A Framework for Analysis of Data Freshness". In: *IQIS 2004, International Workshop on Information Quality in Information Systems, 18 June 2004, Paris, France (SIGMOD 2004 Workshop)*. Ed. by Felix Naumann and Monica Scannapieco, pp. 59–67 (Cited on page 19).

[41] Ming Xiong, Song Han, and Kam-yiu Lam. "A Deferrable Scheduling Algorithm for Real-Time Transactions Maintaining Data Freshness". In: *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS 2005), 6-8 December 2005, Miami, FL, USA* (Cited on page 19).

[42] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*. Vol. 24. Real-Time Systems Series. Springer, 2011 (Cited on pages 19, 22).

[43] Theodore P. Baker. "A Stack-Based Resource Allocation Policy for Realtime Processes". In: *Proceedings of the Real-Time Systems Symposium - 1990, Lake Buena Vista, Florida, USA, December 1990* (Cited on page 21).

[44] R. Rajkumar. “Real-Time Synchronization Protocols for Shared Memory Multiprocessors”. In: *10th International Conference on Distributed Computing Systems (ICDCS 1990), May 28 - June 1, 1990, Paris, France*. IEEE Computer Society, 1990 (Cited on page 22).

[45] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. “Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-Chip”. In: *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*. IEEE Computer Society, 2001, pp. 73–83 (Cited on page 22).

[46] Aaron Block et al. “A Flexible Real-Time Locking Protocol for Multiprocessors”. In: *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007), 21-24 August 2007, Daegu, Korea*. IEEE Computer Society, 2007 (Cited on page 22).

[47] Arvind Easwaran and Björn Andersson. “Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling”. In: 2009 (Cited on page 22).

[48] Farhang Nemati, Moris Behnam, and Thomas Nolte. “Independently-Developed Real-Time Systems on Multi-cores with Shared Resources”. In: 2011 (Cited on page 22).

[49] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. “Real-Time Computing with Lock-Free Shared Objects”. In: *ACM Trans. Comput. Syst.* (1997) (Cited on pages 22, 23, 51).

[50] D. Honegger L. Meier and M. Pollefeys. “PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms”. In: *IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, May 26-30*. 2015 (Cited on pages 23, 42).

[51] Maurice Herlihy. “A methodology for implementing highly concurrent data structures”. In: (1990) (Cited on pages 23, 51).

[52] L. Meier. *PX4 Development Guide*. https://dev.px4.io/master/en/index.html (Cited on pages 23, 39, 43, 50, 65).

[53] Gang Han et al. “Experimental Evaluation and Selection of Data Consistency Mechanisms for Hard Real-Time Applications on Multicore Platforms”. In: *IEEE Trans. Ind. Informatics* (2014) (Cited on page 23).

[54] J. Chen and A. Burns. “Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties.” In: *In proceeding of the International Conference on Real-Time Computing Systems and Applications, pages 236–246*. 1999 (Cited on pages 23–25, 62, 75).

[55] Hai Huang, Padmanabhan Pillai, and Kang G. Shin. “Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems”. In: *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*. Ed. by Carla Schlatter Ellis. 2002 (Cited on pages 23, 24).

[56] Jing Chen and Alan Burns. *A Fully Asynchronous Reader/Writer Mechanism for Multiprocessor Real-Time Systems*. Tech. rep. 1997 (Cited on page 24).

[57] J. Chen and A. Burns. “A three-slot asynchronous reader/writer mechanism for multiprocessor real-time systems”. In: *Technical Report YCS-186, Department of Computer Science, University of York*. 1997 (Cited on page 24).

[58] Hermann Kopetz and Johannes Reisinger. “The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronisation Problem”. In: *Proceedings of the Real-Time Systems Symposium. Raleigh-Durham, NC, USA, December 1993*. 1993 (Cited on pages 24, 25, 62, 63, 75).

[59] Gary L. Peterson. “Concurrent Reading While Writing”. In: *ACM Trans. Program. Lang. Syst.* (1983) (Cited on page 24).

[60] Matthias Becker et al. “End-to-end timing analysis of cause-effect chains in automotive embedded systems”. In: *Journal of Systems Architecture - Embedded Systems Design* 80 (2017), pp. 104–113 (Cited on page 26).

[61] Lothar Michel et al. “Shared SW development in multi-core automotive context”. In: *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. 2016 (Cited on page 28).

[62] Christoph M. Kirsch and Ana Sokolova. “The Logical Execution Time Paradigm”. In: *Advances in Real-Time Systems (to Georg Färber on the occasion of his appointment as Professor Emeritus at TU München after leading the Lehrstuhl für Realzeit-Computersysteme for 34 illustrious years)*. 2012, pp. 103–120 (Cited on pages 28, 54).

[63] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. “Giotto: a time-triggered language for embedded programming”. In: *Proceedings of the IEEE* (2003) (Cited on page 28).

[64] Jorge Martinez, Ignacio Sanudo Olmedo, and Marko Bertogna. “Analytical Characterization of End-to-End Communication Delays With Logical Execution Time”. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* (2018) (Cited on page 28).

[65] Nadège Pontisso. “Association cohérente de données dans les systèmes temps réel à base de composants - Application aux logiciels spatiaux”. PhD thesis. National Polytechnic Institute of Toulouse, France, 2009 (Cited on pages 31, 92).

[66] Nadège Pontisso, Philippe Quéinnec, and Gérard Padiou. “Analysis of distributed multi-periodic systems to achieve consistent data matching”. In: *Concurrency and Computation: Practice and Experience* (2013), pp. 234–249 (Cited on pages 31, 92, 94).

[67] B. Steux et al. “Fade: a vehicle detection and tracking system featuring monocular color vision and radar data fusion”. In: *Intelligent Vehicle Symposium, 2002. IEEE*. 2002 (Cited on pages 31, 92).

[68] Bruno Steux. “RTMAPS, un environnement logiciel dédié à la conception d’applications embarqués tems-réel. Utilisation pour la détection automatique de véhicules par fusion radar/Vision.” PhD thesis. Ecole des mines de Paris, France, 2001 (Cited on pages 31, 80, 92).

Chapter 2: Unmanned aerial vehicles (UAVs)

[7] AUTOSAR. "Spec. of Timing Extensions". In: *AUTOSAR Std. 4.3* (2016) (Cited on pages 8, 9, 27, 28, 55).

[49] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. "Real-Time Computing with Lock-Free Shared Objects". In: *ACM Trans. Comput. Syst.* (1997) (Cited on pages 22, 23, 51).

[50] D. Honegger L. Meier and M. Pollefeys. "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms". In: *IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, May 26-30.* 2015 (Cited on pages 23, 42).

[51] Maurice Herlihy. "A methodology for implementing highly concurrent data structures". In: (1990) (Cited on pages 23, 51).

[52] L. Meier. *PX4 Development Guide.* https://dev.px4.io/master/en/index.html (Cited on pages 23, 39, 43, 50, 65).

[62] Christoph M. Kirsch and Ana Sokolova. "The Logical Execution Time Paradigm". In: *Advances in Real-Time Systems (to Georg Färber on the occasion of his appointment as Professor Emeritus at TU München after leading the Lehrstuhl für Realzeit-Computersysteme for 34 illustrious years).* 2012, pp. 103–120 (Cited on pages 28, 54).

[69] Abdulmalik Humayed et al. "Cyber-physical systems security—A survey". In: *IEEE Internet of Things Journal* 4.6 (2017), pp. 1802–1831 (Cited on page 34).

[70] Ragunathan Rajkumar et al. "Cyber-physical systems: the next computing revolution". In: *Design automation conference.* IEEE. 2010, pp. 731–736 (Cited on page 34).

[71] Torsten Andre et al. "Application-driven design of aerial communication networks". In: *IEEE Communications Magazine* 52.5 (2014), pp. 129–137 (Cited on page 35).

[72] Samira Hayat, Evşen Yanmaz, and Raheeb Muzaffar. “Survey on unmanned aerial vehicle networks for civil applications: A communications viewpoint”. In: *IEEE Communications Surveys & Tutorials* 18.4 (2016), pp. 2624–2661 (Cited on page 36).

[73] Lav Gupta, Raj Jain, and Gabor Vaszkun. “Survey of important issues in UAV communication networks”. In: *IEEE Communications Surveys & Tutorials* 18.2 (2015), pp. 1123–1152 (Cited on page 36).

[74] Satoshi Kohno and Kenji Uchiyama. “Design of robust controller of fixed-wing UAV for transition flight”. In: *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE. 2014, pp. 1111–1116 (Cited on page 36).

[75] Nick Gravish et al. “High-throughput study of flapping wing aerodynamics for biological and robotic applications”. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2014, pp. 3397–3403 (Cited on page 36).

[76] Emad Ebeid, Martin Skriver, and Jie Jin. “A survey on open-source flight control platforms of unmanned aerial vehicle”. In: *2017 Euromicro Conference on Digital System Design (DSD)*. IEEE. 2017, pp. 396–402 (Cited on page 36).

[77] VI Kortunov et al. “Review and comparative analysis of mini-and micro-UAV autopilots”. In: *2015 IEEE international conference actual problems of unmanned aerial vehicles developments (APUAVD)*. IEEE. 2015, pp. 284–289 (Cited on page 36).

[78] 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 (Cited on page 36).

[79] HaiYang Chao, YongCan Cao, and YangQuan Chen. “Autopilots for small unmanned aerial vehicles: a survey”. In: *International Journal of Control, Automation and Systems* 8.1 (2010), pp. 36–44 (Cited on page 36).

[80] Hyon Lim et al. “Build your own quadrotor: Open-source projects on unmanned aerial vehicles”. In: *IEEE Robotics & Automation Magazine* 19.3 (2012), pp. 33–45 (Cited on page 36).

[81] Yibo Li and Shuxi Song. "A survey of control algorithms for quadrotor unmanned helicopter". In: *2012 IEEE fifth international conference on advanced computational intelligence (ICACI)*. IEEE. 2012, pp. 365–369 (Cited on page 37).

[82] Zhicheng Hou et al. "A survey on the formation control of multiple quadrotors". In: *2017 14th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*. IEEE. 2017, pp. 219–225 (Cited on page 37).

[83] MA Saldana-O et al. "A comparative study of the wavenet PID controllers for applications in non-linear systems". In: *2015 12th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*. IEEE. 2015, pp. 1–6 (Cited on page 37).

[84] Vaibhav Unhelkar and Julie Shah. "Contact: Deciding to communicate during time-critical collaborative tasks in unknown, deterministic domains". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. 1. 2016 (Cited on page 38).

[85] Ofra Amir, Barbara Grosz, and Roni Stern. "To share or not to share? the single agent in a team decision problem". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 28. 1. 2014 (Cited on page 38).

[86] Maayan Roth, Reid Simmons, and Manuela Veloso. "Reasoning about joint beliefs for execution-time communication decisions". In: *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. 2005, pp. 786–793 (Cited on page 38).

[87] Maayan Roth, Reid Simmons, and Manuela Veloso. "What to communicate? Execution-time decision in multi-agent POMDPs". In: *Distributed autonomous robotic systems 7*. Springer, 2006, pp. 177–186 (Cited on page 38).

[88] Chongjie Zhang and Victor Lesser. "Coordinating multi-agent reinforcement learning with limited communication". In: *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*. 2013, pp. 1101–1108 (Cited on page 38).

[89] Pranav Bhatkhande and Timothy C Havens. “Real time fuzzy controller for quadrotor stability control”. In: *2014 IEEE international conference on fuzzy systems (FUZZ-IEEE)*. IEEE. 2014, pp. 913–919 (Cited on page 38).

[90] Bora Erginer and Erdinç Altuğ. “Design and implementation of a hybrid fuzzy logic controller for a quadrotor VTOL vehicle”. In: *International Journal of Control, Automation and Systems* 10.1 (2012), pp. 61–70 (Cited on page 38).

[91] Theerasak Sangyam et al. “Path tracking of UAV using self-tuning PID controller based on fuzzy logic”. In: *Proceedings of SICE annual conference 2010*. IEEE. 2010, pp. 1265–1269 (Cited on page 38).

[92] Ya Wang, Hongbin Zhang, and Dongfei Han. “Neural network adaptive inverse model control method for quadrotor UAV”. In: *2016 35th Chinese Control Conference (CCC)*. IEEE. 2016, pp. 3653–3658 (Cited on page 38).

[93] Steven Lake Waslander et al. “Multi-agent quadrotor testbed control design: Integral sliding mode vs. reinforcement learning”. In: *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2005, pp. 3712–3717 (Cited on page 38).

[94] Haitham Bou-Ammar, Holger Voos, and Wolfgang Ertel. “Controller design for quadrotor uavs using reinforcement learning”. In: *2010 IEEE International Conference on Control Applications*. IEEE. 2010, pp. 2130–2135 (Cited on page 38).

[95] Ivana Palunko et al. “A reinforcement learning approach towards autonomous suspended load manipulation using aerial robots”. In: *2013 IEEE international conference on robotics and automation*. IEEE. 2013, pp. 4896–4901 (Cited on page 38).

[96] Xingguang Peng, Demin Xu, and Xiaoguang Gao. “A dynamic genetic algorithm based on particle filter for UCAV formation control”. In: *Proceedings of the 29th Chinese Control Conference*. IEEE. 2010, pp. 5238–5241 (Cited on page 38).

[97] Travis Dierks and Sarangapani Jagannathan. “Neural network control of quadrotor UAV formations”. In: *2009 American Control Conference*. IEEE. 2009, pp. 2990–2996 (Cited on page 38).

[98] Bart Van Arem, Cornelie JG Van Driel, and Ruben Visser. "The impact of cooperative adaptive cruise control on traffic-flow characteristics". In: *IEEE Transactions on intelligent transportation systems* 7.4 (2006), pp. 429–436 (Cited on page 39).

[99] Victor Casas and Andreas Mitschele-Thiel. "On the impact of communication delays on UAVs flocking behavior". In: *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*. IEEE. 2018, pp. 67–72 (Cited on page 39).

[100] Xinyi Liu et al. "UD-MAC: Delay tolerant multiple access control protocol for unmanned aerial vehicle networks". In: *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. IEEE. 2017, pp. 1–6 (Cited on page 39).

[101] Yoonsoo Kim and Mehran Mesbahi. "On maximizing the second smallest eigenvalue of a state-dependent graph Laplacian". In: *Proceedings of the 2005, American Control Conference, 2005.* IEEE. 2005, pp. 99–103 (Cited on page 39).

[102] L. Meier. *The MAVLink messaging.* http://dev.px4.io/master/en/middleware/mavlink.html (Cited on page 42).

[103] L. Meier. *RTPS/ROS2 Interface: PX4-FastRTPS Bridge.* http://dev.px4.io/master/en/middleware/micrortps.html (Cited on page 42).

[104] Patrick Th. Eugster et al. "The many faces of publish/subscribe". In: *ACM Comput. Surv.* (2003) (Cited on page 42).

[105] Gan Deng et al. "Evaluating Real-Time Publish/Subscribe Service Integration Approaches in QoS-Enabled Component Middleware". In: *Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), 7-9 May 2007, Santorini Island, Greece* (Cited on page 42).

[106] Heithem Abbes et al. "Performance Analysis of Publish/Subscribe Systems". In: (2007) (Cited on page 42).

[107] Stavros Tripakis. "Description and Schedulability Analysis of the Software Architecture of an Automated Vehicle Control System". In: *Proceedings of the Second Inter-*

national Conference on Embedded Software. EMSOFT '02. 2002, pp. 123–137 (Cited on pages 42, 43).

Chapter 3: Modeling and formalization

[2] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* (1973) (Cited on pages 5, 9, 16, 60, 61, 64, 103).

[10] Joseph Y.-T. Leung and Jennifer Whitehead. "On the complexity of fixed-priority scheduling of periodic, real-time tasks". In: *Perform. Evaluation* (1982) (Cited on pages 9, 60).

[52] L. Meier. *PX4 Development Guide*. https://dev.px4.io/master/en/index.html (Cited on pages 23, 39, 43, 50, 65).

[54] J. Chen and A. Burns. "Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties." In: *In proceeding of the International Conference on Real-Time Computing Systems and Applications, pages 236–246*. 1999 (Cited on pages 23–25, 62, 75).

[58] Hermann Kopetz and Johannes Reisinger. "The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronisation Problem". In: *Proceedings of the Real-Time Systems Symposium. Raleigh-Durham, NC, USA, December 1993*. 1993 (Cited on pages 24, 25, 62, 63, 75).

[108] Falou Ndoye and Yves Sorel. "Monoprocessor Real-Time Scheduling of Data Dependent Tasks with Exact Preemption Cost for Embedded Systems". In: 2013 (Cited on page 60).

[109] Falou Ndoye. "Ordonnancement temps reel preemptif multiprocesseur avec prise en compte du cout du systeme d'exploitation." PhD thesis. PhD thesis, Universite de Paris Sud, 2014 (Cited on page 60).

[110] EmbedJournal. *Implementing Circular Buffer in C*. https://embedjournal.com/implementing-circular-buffer-embedded-c/ (Cited on page 63).

[111] Hermann Kopetz et al. “Distributed fault-tolerant real-time systems: the Mars approach”. In: *IEEE Micro* (1989) (Cited on page 63).

[112] Jorgen Bang-Jensen. “2.1 Acyclic Digraphs”. In: *Digraphs: Theory, Algorithms and Applications, Springer Monographs in Mathematics (2nd ed.), Springer-Verlag* (2008) (Cited on page 69).

[113] Thulasiraman K. and Swamy M. N. S. “5.7 Acyclic Directed Graphs”. In: *Graphs: Theory and Algorithms, John Wiley and Son* (1992) (Cited on page 69).

[114] Asratian Armen S., Denley Tristan M. J.and Haggkvist, and Roland. “Bipartite Graphs and their Applications”. In: *Cambridge Tracts in Mathematics, 131, Cambridge University Press* (1998) (Cited on page 69).

Chapter 4: Local consistency constraints

[5] M. Joseph and P. Pandya. “Finding Response Times in a Real-Time System”. In: *Comput. J.* (1986) (Cited on pages 6, 14, 16, 77).

[54] J. Chen and A. Burns. “Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties.” In: *In proceeding of the International Conference on Real-Time Computing Systems and Applications, pages 236–246*. 1999 (Cited on pages 23–25, 62, 75).

[58] Hermann Kopetz and Johannes Reisinger. “The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronisation Problem”. In: *Proceedings of the Real-Time Systems Symposium. Raleigh-Durham, NC, USA, December 1993*. 1993 (Cited on pages 24, 25, 62, 63, 75).

[68] Bruno Steux. “RTMAPS, un environnement logiciel dédié à la conception d’applications embarqués tems-réel. Utilisation pour la détection automatique de véhicules par fusion radar/Vision.” PhD thesis. Ecole des mines de Paris, France, 2001 (Cited on pages 31, 80, 92).

Chapter 5: Global consistency constraints

[2] C. L. Liu and James W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. In: *J. ACM* (1973) (Cited on pages 5, 9, 16, 60, 61, 64, 103).

[31] S Mubeena M Becker D Dasari, M Behnam, and Thomas Nolte. “Synthesizing Job-Level Dependencies for Automotive Multi-Rate Effect Chains.” In: *RTCSA, At Daegu, South Korea*. 2016 (Cited on pages 17, 114).

[65] Nadège Pontisso. “Association cohérente de données dans les systèmes temps réel à base de composants - Application aux logiciels spatiaux”. PhD thesis. National Polytechnic Institute of Toulouse, France, 2009 (Cited on pages 31, 92).

[66] Nadège Pontisso, Philippe Quéinnec, and Gérard Padiou. “Analysis of distributed multi-periodic systems to achieve consistent data matching”. In: *Concurrency and Computation: Practice and Experience* (2013), pp. 234–249 (Cited on pages 31, 92, 94).

[67] B. Steux et al. “Fade: a vehicle detection and tracking system featuring monocular color vision and radar data fusion”. In: *Intelligent Vehicle Symposium, 2002. IEEE*. 2002 (Cited on pages 31, 92).

[68] Bruno Steux. “RTMAPS, un environnement logiciel dédié à la conception d’applications embarqués tems-réel. Utilisation pour la détection automatique de véhicules par fusion radar/Vision.” PhD thesis. Ecole des mines de Paris, France, 2001 (Cited on pages 31, 80, 92).