Modeling and scheduling embedded real-time systems using Synchronous Data Flow Graphs

Jad Khatib

THÈSE DE DOCTORAT DE
SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique
(Paris)

Présentée par

Jad KHATIB

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse :

Modélisation et ordonnancement des systèmes temps réel embarqués utilisant des graphes de flots de données synchrones

Modeling and scheduling embedded real-time systems
using Synchronous Data Flow Graphs

soutenue le 11 septembre 2018

devant le jury composé de :

Mme. Alix Munier-Kordon	Directrice de thèse
Mme. Kods Trabelsi	Encadrante de thèse
M. Frédéric Boniol	Rapporteur
M. Pascal Richard	Rapporteur
M. Laurent George	Examinateur
M. Lionel Lacassagne	Examinateur
M. Dumitru Potop-Butucaru	Examinateur
M. Etienne Borde	Examinateur

Acknowledgements

I would like to take the opportunity to thank all those who supported me during my thesis.

I have an immense respect, appreciation and gratitude towards my advisors: Ms. Alix Munier-Kordon, Professor at Sorbonne Université, and Ms. Kods Trabelsi, Research Engineer at CEA LIST, for their consistent guidance and support during my thesis journey. I would like to thank them for sharing their invaluable knowledge, experience and advice which have been greatly beneficial for the successful completion of this thesis.

I would like to extend my sincere gratitude to the reviewers, Professor Frédéric Boniol and Professor Pascal Richard for accepting to evaluate my thesis. My earnest thanks are also due to Professor Laurent George, Professor Lionel Lacassagne, Doctor Dumitru Potop-Butucaru and Doctor Etienne Borde for accepting to be a member of the jury.

Thanks to all Lastre and LCE team members for the occasional coffees and discussions especially during our "PhD Days". Thanks also to my colleagues in Lip6 and Télécom Paristech: Youen, Cédric, Eric, Olivier, Roberto and Kristen as well as all those I forgot to mention.

I would also like to thank my friends Sarah, Laure, Marcelino, Hadi, Serge and Damien for their support and all our happy get-togethers and for uplifting my spirit during the challenging times faced during this thesis.

I am particularly grateful to my friend Catherine for her priceless friendship and precious support and for continuously giving me the courage to make this thesis better.

Last but definitely not least, I am forever in debt to my awesomely supportive mother Zeina and my wonderful sister Jihane for inspiring and guiding me throughout the course of my life and hence this thesis.

Résumé

Les systèmes embarqués temps réel impactent nos vies au quotidien. Leur complexité s'intensifie avec la diversité des applications et l'évolution des architectures des plateformes de calcul. En effet, les systèmes temps réel peuvent se retrouver dans des systèmes autonomes, comme dans les métros, avions et voitures autonomes. Ils sont donc souvent d'une importance décisive pour la vie humaine, et leur dysfonctionnement peut avoir des conséquences catastrophiques. Ces systèmes sont généralement multi-périodiques car leurs composants interagissent entre eux à des rythmes différents, ce qui rajoute de la complexité. Par conséquent, l'un des principaux défis auxquels sont confrontés les chercheurs et industriels est l'utilisation de matériel de plus en plus complexe, dans le but d' exécuter des applications temps réel avec une performance optimale et en satisfaisant les contraintes temporelles. Dans ce contexte, notre étude se focalise sur la modélisation et l'ordonnancement des systèmes temps réel en utilisant un formalisme de flot de données. Notre contribution a porté sur trois axes:

Premièrement, nous définissons un mode de communication général et intuitif au sein de systèmes multi-périodiques. Nous montrons que les communications entre les tâches multi-périodiques peuvent s'exprimer directement sous la forme d'une classe spécifique du "Synchronous Data Flow Graph" (SDFG). La taille de ce graphe est égale à la taille du graphe de communication. De plus, le modèle SDFG est un outil d'analyse qui repose sur de solides bases mathématiques, fournissant ainsi un bon compromis entre l'expressivité et l'analyse des applications.

Deuxièmement, le modèle SDFG nous a permis d'obtenir une définition précise de la latence. Par conséquent, nous exprimons la latence entre deux tâches communicantes à l'aide d'une formule close. Dans le cas général, nous développons une méthode d'évaluation exacte qui permet de calculer la latence du système dans le pire des cas. Ensuite, nous bornons la valeur de la latence en utilisant deux algorithmes pour calculer les bornes inférieure et supérieure. Enfin, nous démontrons que les bornes de la latence peuvent être calculées en temps polynomial, alors que le temps nécessaire pour évaluer sa valeur exacte augmente linéairement en fonction du facteur de répétition moyen.

Finalement, nous abordons le problème d'ordonnancement mono-processeur des systèmes strictement périodiques non-préemptifs, soumis à des contraintes de com-

munication. En se basant sur les résultats théoriques du SDFG, nous proposons un algorithme optimal en utilisant un programme linéaire en nombre entier (PLNE). Le problème d'ordonnancement est connu pour être NP-complet au sens fort. Afin de résoudre ce problème, nous proposons trois heuristiques: relaxation linéaire, simple et ACAP. Pour les deux dernières heuristiques, et dans le cas où aucune solution faisable n'est trouvée, une solution partielle est calculée.

Mots clés: systèmes temps réel, graphes de flots de données synchrones, modélisation des communications, tâches strictement périodiques non-préemptifs, ordonnancement mono-processeur, latence.

Abstract

Real-time embedded systems change our lives on a daily basis. Their complexity is increasing with the diversity of their applications and the improvements in processor architectures. These systems are usually multi-periodic, since their components communicate with each other at different rates. Real-time systems are often critical to human lives, their malfunctioning could lead to catastrophic consequences. Therefore, one of the major challenges faced by academic and industrial communities is the efficient use of powerful and complex platforms, to provide optimal performance and meet the time constraints. Real-time system can be found in autonomous systems, such as air-planes, self-driving cars and drones. In this context, our study focuses on modeling and scheduling critical real-time systems using data flow formalisms. The contributions of this thesis are threefold:

First, we define a general and intuitive communication model within multi-periodic systems. We demonstrate that the communications between multi-periodic tasks can be directly expressed as a particular class of "Synchronous Data Flow Graph" (SDFG). The size of this latter is equal to the communication graph size. Moreover, the SDFG model has strong mathematical background and software analysis tools which provide a compromise between the application expressiveness and analyses.

Then, the SDFG model allows precise definition of the latency. Accordingly, we express the latency between two communicating tasks using a closed formula. In the general case, we develop an exact evaluation method to calculate the worst case system latency from a given input to a connected outcome. Then, we frame this value using two algorithms that compute its upper and lower bounds. Finally, we show that these bounds can be computed using a polynomial amount of computation time, while the time required to compute the exact value increases linearly according to the average repetition factor.

Finally, we address the mono-processor scheduling problem of non-preemtive strictly periodic systems subject to communication constraints. Based on the SDFG theoretical results, we propose an optimal algorithm using MILP formulations. The scheduling problem is known to be NP-complete in the strong sense. In order to solve this issue, we proposed three heuristics: linear programming relaxation, *simple* and ACAP heuristics. For the second and the third heuristic if no feasible solution is found, a partial solution is computed.

Keywords: real-time systems, Synchronous Data Flow Graph, communications modeling, non-preemptive strictly periodic tasks, mono-processor scheduling, latency.

Table of contents

List of figures

List of tables

Table of Acronyms

ACAP	As Close As Possible
ADL	Architecture Description Language
API	Application Programming Interface
ASAP	As Soon As Possible
CAN	Controller Area Network
CP	Computation Graph
CPU	Central Processing Unit
CSDFG	Cyclo-Static Dataflow Graph
DAG	Directed Acyclic Graph
DM	Deadline Monotonic
DP	Dynamic Priority
DPN	Dataflow Process Network
EDF	Earliest Deadline First
E/E	Electric/Electronic
FIFO	First-In-First-Out
GCD	Greatest Common Divisor
HSDFG	Homogeneous Synchronous Data Flow
ILP	Integer Linear Programming

I/O	Input/Output
KPN	Kahn Process Network
LCM	Least Common Multiple
LET	Logical Execution Time
LP	Linear Programming
MILP	Mixed Integer Linear Programming
MPPA	Massively Parallel Processor Array
NoC	Network on Chip
OS	Operating System
PGM	Processing Graphs Methods
RBE	Rate Base Execution
RM	Rate Monotonic
RTA	Response Time Analysis
RTOS	Real-Time Operating System
RTSDFG	Real-Time Synchronous Data Flow Graph
SBD	Synchronous Block Diagrams
SDFG	Synchronous Data Flow Graph
SWC	Software Component
TTA	Time Triggered Architectures
WCET	Worst Case Execution Time
WCRT	Worst Case Response Time

Chapter 1

Introduction

Nowadays, electronic devices are ubiquitous in our daily lives. Every time we set an alarm, drive a car, take a picture or use a cell-phone, we interact in some way with electronic components or devices. Electronic systems make people's lives safer, faster, easier and more convenient. These systems, integrated in a larger environment with specific assignments or purposes, are called **embedded systems**. They comprise a hardware part (electronic) that defines interfaces and system performance as well as a software part (computer) that dictates their functions. Such systems are autonomous and limited in size, power consumption, and heat dissipation. In order to achieve specific tasks, their design combines skills from computer and electronics fields.

An important class of embedded systems is **real-time embedded systems**. They operate dynamically in their environments and must continuously adapt to their changes. Indeed, they fully command their environments through actuators upon data reception via sensors. The correctness of such systems depends not only on the logical result but also on the time it is delivered. Therefore, real-time systems are classified according to the temporal constraints severity: soft, firm and hard (critical) systems. In hard systems, the violation of a temporal constraint can lead to catastrophic consequences. The aircraft piloting system, the control of a nuclear power plant and the track control system are typical examples. The soft and firm systems are more "tolerant": the temporal constraint violation does not cause serious damages to the environment and does not corrupt the system behaviour. In fact, this violation leads to a degradation of the result quality (i.e performance).

On the other hand, critical real-time systems are becoming increasingly complex. They require research effort in order to be effectively modeled and executed. In this context, one of the major challenges faced by academic and industrial environments is the efficient use of powerful and complex resources, to provide optimal performance

and meet the time constraints. These systems are usually multi-periodic, since their components communicate with each other at different rates. This is mainly due to their physical characteristics. For this reason, a deep analysis of communications within multi-periodic systems is required. This analysis is essential to schedule these systems on a given platform. Moreover, the estimation of parameters in a static manner, such as evaluating the latency between the system input and its corresponding outcomes, is an important practical issue.

1.1 Contributions

In this section, we summarize the research contribution of this thesis.

The first contribution consists in defining a general and intuitive communication model for multi-periodic systems. Based on the precedence constraints between the tasks executions, we demonstrate that the communications between multi-periodic tasks can be directly expressed as a "Synchronous Data-flow Graph". The size of this graph is equal to the application size. We called this particular class "Real-Time Synchronous Data-flow Graphs". These results were published in a short paper in "Ecole d'été temps réel"(ETR 2015). In collaboration with Cedric Klikpo (projet ELA, IRT SystemX), we showed that the communications of an application expressed with Simulink can be modeled by a SDFG. This transformation led to an article published in RTAS 2016 [76].

The second contribution consists in evaluating the worst-case latency of multi-periodic systems. Based on our communication model, we define and compute the latency between two communicating tasks. We prove that minimum and maximum latency between the tasks executions can be computed according to the tasks parameters using closed formulas. Moreover, we bounded their values according to the tasks periods. In order to evaluate the worst-case system latency, we propose an exact pricing algorithm. This method computes the system latency in terms of the average repetition factor. This implies that if this factor is not bounded, the complexity of the exact evaluation increases exponentially. Consequently, we propose two polynomial-time algorithms that compute respectively the upper and lower bounds of this latency. This evaluation led to an article published in RTNS 2016 [73].

The third contribution consists in solving the mono-processor scheduling problem of non-preemptive strictly periodic systems subject to communication constraints. Based

on our communication model, we proved that scheduling two periodic communicating tasks is equivalent to schedule two independent tasks. These tasks can only be executed between their release dates and their deadlines. Accordingly, we propose an optimal algorithm using "Mixed Integer Linear Programming" formulations. We study two case: the *fixed* and *flexible* interval cases. As the scheduling problem in known to be NP-complete in strong sense [79], we propose three heuristics: linear programming relaxation, *simple* and ACAP heuristics. For the second and the third heuristic if no feasible solution is found, a partial solution is computed. This solution contains the subset of tasks that can be executed on the same processor.

1.2 Thesis overview

The thesis is organized as follows.

Chapter 2 introduces the context of this thesis. First, it presents the real-time computational model and its characteristics. Afterwards, it formulates the first two problems addressed in this thesis: modeling communications within multi-periodic systems and evaluating their latencies. This chapter reviewed several existing approaches regarding multi-periodic systems modeling. Finally, basic notions of real-time scheduling are presented in order to formulate the third problem studied in this thesis: mono-processor scheduling of non-preemptive strictly periodic set of tasks with communication constraints.

Chapter 3 presents an overview of static data flow models. Important notions and transformations are introduced in the context of the Synchronous Data Flow Graph: precedence constraints, consistency, repetition vector, normalization and expansion.

Chapter 4 gives an overview on the state of the art related to this thesis. It positions our study regarding existing approaches for the two aspects:

- modeling real-time systems and evaluating their latencies using data formalisms,
- scheduling non-preemptive strictly periodic systems.

Chapter 5 introduces the first two contributions of this thesis. It defines the "Real-Time Synchronous Data Flow Graph" which models the communications within a multi-

periodic system. On the other hand, this chapter describes the latency computation method between two periodic communicating tasks. This latency is computed according to the tasks parameters with a closed formula and its value is bounded according to the tasks periods. Several methods are described for evaluating the worst case system latency: exact evaluation, upper and lower bounds.

Chapter 6 introduces the third contribution of this thesis. It presents the mono-processor scheduling problem of non-preemptive strictly periodic systems subject to communication constraints. In order to solve the scheduling problem, an exact method is developed using the "Mixed Integer Linear Programming" formulations. Two cases are treated: the *fixed* and *flexible* interval cases. In the *fixed* case, tasks execution is restricted to the interval between its release date and its deadline. However, in the *flexible* case, tasks can admit new execution intervals whose lengths are equal to their relative deadlines. Furthermore, several heuristics are proposed: linear programming relaxation, *simple* and ACAP heuristics.

Chapter 7 presents the experimental results of this thesis. It is devoted to evaluate the methods that compute the worst system latency using the RTSDFG formalism. In addition, it presents several experiments dedicated to evaluate the performance of our methods that solve the mono-processor scheduling problem.

Chapter 8 concludes this thesis and introduces some perspectives for future work.

Chapter 2

Context and Problems

Embedded real-time systems are omnipresent in our daily life, they cover a range of different levels of complexity. We find Embedded real-time systems in several domains: robotics, automotive, aeronautics, medical technology, telecommunications, railway transport, multimedia and nuclear power plants. These systems are composed of hardware and software devices with functional and timing constraints. In fact, the correctness of such systems depends not only on the logical result but also on the physical time at which this result is produced. Real-time systems are usually reactive, since they interact permanently with their environments. In addition, embedded real-time systems are often considered *critical*. This is due to the fact that the computer system is entrusted with a great responsibility in terms of human lives and non-negligible economic interests. In order to predict the behavior of an embedded safety-critical system, we need to know the interactions between its components and how to schedule these components on a given platform.

This chapter introduces the required background to understand the thesis contributions presented in the following chapters. The remainder of this chapter is organized as follows. Section 2.1 introduces the real-time computational model and its characteristics. Section 2.2 formulates the first two problems addressed in this thesis: modeling communications within multi-periodic systems and evaluating their latencies. Section 2.3 presents several approaches from the literature dedicated to model multi-periodic applications. Section 2.4 presents some basic notions of the scheduling problem. Section 2.5 formulates the third problem studied in this thesis: mono-processor scheduling problem of non-preemptive strictly periodic systems subject to communication constrains. Finally, Section 2.6 concludes the chapter.

2.1 Real-time systems

In this section, we introduce the real-time systems and their characteristics. We present the task models on which the approaches developed in Chapters 5 and 6 are applied.

2.1.1 Definition

There are several definitions of real-time systems in the literature [22, 111, 39]. J.Stankovic [111] proposed a functional definition as follows:

"A real-time computer system is defined as a system whose correctness depends not only on the logical results of computations, but also on the physical time at which the results are produced."

In other words, a real-time system should satisfy two types of constraints:

- logical constraints corresponding to the computation of the system's output according to its input.
- temporal constraints corresponding to the computation of the system's output within a time frame specified by the application. A delayed production of the system's output considered as an error which can lead to serious consequences.

J-P.Elloy [39] defines a real-time system in an operational way:

"A real-time system is defined as any application implementing a computer system whose behaviour is conditioned by the state dynamic evolution of the process which is assigned to it. This computer system is then responsible for monitoring or controlling this process while respecting the application temporal constraints."

This latter definition clarifies the meaning of the term "real-time" by altering the relationships between the computer systems and external environment.

The real-time systems are *reactive* systems [58, 15]. Their primary purpose is to continuously react to stimulus from their environment which are considered external to the system. A formal definition of a reactive system, describing the functioning of a real-time system, was given in [47]:

"A reactive system is a system that reacts continuously with its environment at a rate imposed by this environment. It receives inputs coming from the environment called stimuli via sensors. It reacts to all these stimuli by performing a certain number of operations and. It produces through actuators outputs that will be used by the environment. These outputs are called reactions or commands."

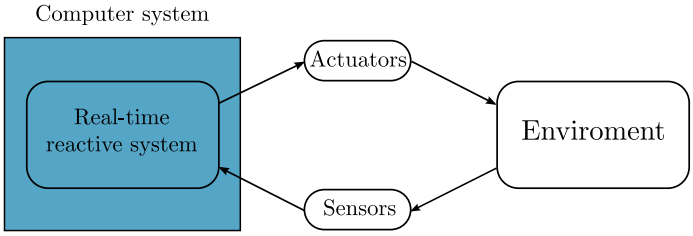


Fig. 2.1 Schematic representation of a real-time system.

Usually, a real-time system must react to each stimuli coming from its environments. In addition, the system response depends not only on the input stimuli but on the system state at the moment when the stimuli arrives. The interface between a real-time system and its environment consists of two types of peripheral devices. *Sensors* are input devices used to collect a flow of information emitted by the environment. *Actuators* are output devices which provide the environment with control system commands (see Fig. 2.1). A real-time embedded system is an integrated system inside the controlled environment, such as a calculator in a car.

2.1.2 Real-Time systems classification

The validity of real-time systems is related to the result quality (range of accepted values) and the limited duration of the result computation (deadline). Real-time systems can be classified from different perspectives [78]. These classifications depend on the characteristics of the application (factors outside the computer system) or on the design and implementation (factors inside the computer system). According to the temporal constraints criticality, there are three types of real-time systems:

Hard real-time systems

The hard (critical) real-time systems consist of processings that have strict temporal constraints. A hard real-time system requires that all the system processing must imperatively respect all the temporal constraints. In these systems, the violation of

a constraint can lead to catastrophic consequences. The aircraft piloting system, the control of a nuclear power plant and the track control system are typical examples. To ensure the proper functioning of a real-time system, we must be able to define conditions on the system environment. Under subsequent conditions, it is also necessary to guarantee the temporal constraints for all possible execution scenarios.

Soft real-time systems

The soft real-time systems are more "tolerant". This means that these systems are less demanding in terms of respecting the temporal constraints than the hard systems. The constraint violation does not cause serious damages to the environment and does not corrupt system behaviour. In fact, this violation leads to a degradation of the result quality (performance). This is the case of multimedia applications such as image processing, where it is acceptable to have a precise number of images (image processing) with a sound shift of few milliseconds. One problem of these systems is to evaluate their performance while respecting the quality of service constraints [64].

Firm real-time systems

The Firm real-time systems are composed of processing with strict and soft temporal constraints. These systems [14] can tolerate a clearly specified degree of missed deadlines. This means that deadlines can be missed occasionally by providing a late worthless result. The extent to which a system may tolerate missed deadlines has to be stated precisely.

In this thesis we are interested only in hard real-time systems such as motor drive applications in cars, buses and trucks.

2.1.3 Real-time tasks' characteristics

A real-time task is a set of instructions intended to be executed on a processor. It can be executed several times during the lifetime of the real-time system. For example, an input task that responds to the informations coming from a sensor. Each task execution is called an instance or job. In order to meet the real-time system temporal requirements, time constraints are defined for each real-time task (see Figure 2.2) of an application. The most used parameters of a real-time task t_i are:

- **Activation period T_i:** the minimum delay between two consecutive activations of t_i is the task period. A task is periodic if the task instances are activated regularly at a constant rate.

- **Release date r_i:** the earliest date on which the task can begin its execution.
- **Start date s_i:** it is also called the start execution date. It is the date on which the task starts its execution on a processor.
- **End date e_i:** the date on which the task ends its execution on a processor.
- **Execution time C_i:** the required time for a processor to execute an instance of a task t_i. In general, this parameter is evaluated as the task worst case execution time (WCET) on a processor. The WCET represents an upper bound of the task execution time. The value of this parameter should not be overestimated. The problem of estimating the task execution time has been extensively studied in the literature [104, 106, 99, 113].
- **Deadline D_i:** it corresponds to the date at which the task must complete its execution. Exceeding the due date (deadline) causes a violation of the temporal constraints. we distinguish two types of deadlines:
  - Relative deadline: D_i the time interval between the task release date and its absolute deadline.
  - Absolute deadline: the date at which the task must be completed $(r_i + D_i)$.

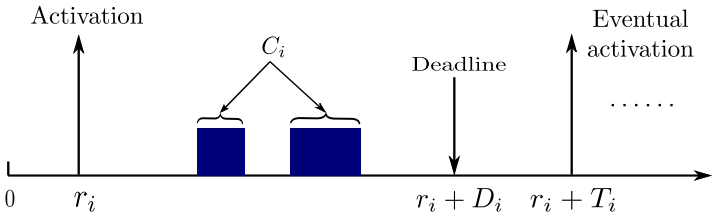


Fig. 2.2 Real-time tasks' characteristics.

Some parameters are derived from the basic parameters such as the **utilization factor**:

$$U_i = \frac{C_i}{T_i}.$$

Dynamic parameters are used to follow the behaviour of the task executions:

- r_i^k : the release date of the kth task instance (t_i^k). In the periodic case, this date can be calculated according to the task first release date r_i^1 :

$$r_i^k = r_i^1 + (k-1) \cdot T_i$$

- s_i^k : the start date of the kth task instance (t_i^k).
- e_i^k : the end date of the kth task instance (t_i^k).
- d_i^k : the absolute deadline of the kth task instance (t_i^k). In the periodic case, this date can be computed in function of the relative deadline (D_i) and the kth release date r_i^k :

$$d_i^k = r_i^k + D_i = r_i^1 + (k-1) \cdot T_i + D_i.$$

- R_i^k : the response time of the kth task instance (t_i^k). It is equal to $e_i^k - r_i^k$.

2.1.4 Tasks models

The real-time system environment defines the system temporal constraints. In other words, it defines the activation dates pace of the tasks. A task can be activated randomly (non-periodic) or at regular intervals (periodic). There are three classes of real-time tasks:

Periodic tasks

Tasks with regular arrival times T_i are called periodic tasks. A common use of periodic tasks is to process sensor data and update the current state of the real-time system on a regular basis. Periodic tasks usually have hard deadlines, but in some applications the deadlines can be soft. This class is based on the model of Liu and Layand [85].

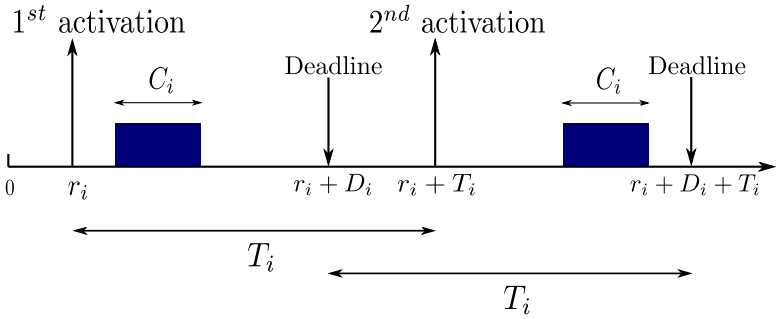


Fig. 2.3 Liu and Layland model.

An example of Liu and Layland model is depicted in Figure 2.3. Each task t_i is characterized by an activation period T_i, an execution time C_i (all instances of a periodic task have the same worst case execution time C_i), a deadline D_i, and eventually

a release date (date of the first activation) r_i. Each task instance must be executed entirely within the interval of length T_i. Successive executions of t_i , admit periodic release dates and deadlines, where the period is equal to T_i.

Strictly periodic tasks is a particular case of this class. In addition to the Liu and Layland model, the task successive executions admit periodic start dates. In other words, the first instance (job) t_i^1 of task t_i, which is released at r_i^1, starts its execution at time s_i^1. Then, in every following period of t_i, t_i^k is released at r_i^k and starts its execution at s_i^k. The kth release and start dates of t_i can be written as follows:

$$r_i^k = r_i^1 + (k-1) \cdot T_i$$
$$s_i^k = s_i^1 + (k-1) \cdot T_i$$

In the following of the manuscript, r_i^1 and s_i^1 are noted r_i and s_i respectively.

Aperiodic tasks

An aperiodic task is a stream of instances arriving at irregular intervals. Aperiodic task has a relative deadline, with no activation dates and no activation periods. Thus, the task activation dates are random and can not be anticipated. These activations are determined by the arrival of events that can be triggered at any time such as message from an operator. Several processes deal with this class of tasks as shown in [110].

Sporadic tasks

A sporadic task is an aperiodic task with a hard deadline and a minimum period p_i [94]. Two successive instances of a task must be separated by at least p_i time units. Note that without a minimum inter-arrival time restriction, it is impossible to guarantee that a sporadic task's deadline would always be met.

2.1.5 Execution mode

Multiple instances of a task can be executed during the lifetime of a real-time system. We distinguish two executions modes:

Non-preemptive mode

In non-preemptive execution mode, the scheduler cannot interrupt the task execution in favour of another one. In order to execute a new task, it is necessary to wait until

the end of the current task. In other words, a task must voluntarily give it up the control of the processor before the execution of an other task.

Preemptive mode

The difference between the non-preemptive execution mode and the preemptive one is that the later gives the scheduler the control of the processor without the task's cooperation. According to the scheduling algorithm, the currently running task loses control of the processor when a task with higher priority is ready to be executed regardless of whether it has finished its execution or not.

In this thesis, we focus our study on modeling and scheduling applications composed of non-preempting strictly periodic tasks.

2.1.6 Communication constraints

Designers describe the embedded real-time systems workflow using block diagrams. These blocks correspond to functions that may be independent or dependent. Functions are considered tasks as soon as they have been characterized temporally. Most of real-time applications require communication between tasks. This type of communication connects an emitting task to a receiving one. The emitting task produces data that are consumed by the receiving task through a communication point. However, a receiving task cannot consume a piece of data unless this data has been sent by the emitting task. Thus, the execution of a receiving task should be preceded by the execution of an emitting task, which imposes precedence constraints between some executions. It should be noted that the precedence constraint between executions (jobs) is mostly due to the fact that both tasks are dependent. The dependency between the tasks can be described by a directed graph. The graph nodes represent the tasks and its arcs represent the dependency relations between the tasks. There are two types of dependencies:

- Dependencies that involve a loss of data. In this case, the receiving task can not consume all the data produced by the emitting one. These data (which are not consumed) have been overwritten by the newer data which are produced by more recent emitting task executions. This is the case where the periods of dependent tasks correspond to prime numbers.
- Dependencies that do not involve a loss of data. In this case, the data produced by the emitting task executions are all consumed by the receiving task executions.

In this thesis, we consider a communication mechanism between tasks which may involve a loss of data. In other words, a task can be executed using the most recent data of its predecessors. The description of the communication scheme between multi-periodic tasks will be detailed in section 5.1 of chapter 5

2.1.7 Latency

Latency constraint between two tasks t_i and t_j is equivalent to impose that the gap between the end date of t_j and the start date of t_i does not exceed a certain value $\mathcal{L}$. Limiting the time gap between these tasks guarantees that the response time - the total time required by a task in order accomplish its execution - of the second task will never exceed a preset critical value. Exceeding this value may lead to performance degradation as well as instability of the system.

System latency evaluation

Real-time systems have to provide results to its environment in a timely fashion. This requirement is typically measured by the duration (time gap) between an input event arriving at the system input and its corresponding result events coming out from the system output. This time gap is denoted as the system latency. Executing a system with a long response time (fairly significant latency) may cause a lot of damages, such as the late identification of an obstacle in an Advanced Driver Assistance Systems [117]. Depending on the application scenario, the system latency can be evaluated with different accuracy classes: worst-case latency, best-case latency, average latency or different quantiles of latency.

In this thesis, we are interested in evaluating the worst-case system latency.

2.1.8 Multi-periodic systems

Complex real-time embedded systems should handle multiple distinct periods. This is mainly induced by the physical characteristic of sensors and actuators. A multi-periodic system is composed of a set of periodic tasks communicating with each other and having distinct periods. Each task is executed according to its own period. However, the entire system is executed according to a global period which is called the "hyper-period"(hp) [50]. This global period is equal to the Least Common Multiple (LCM) of all the system tasks periods. A task's execution is called a Job or instance. The number of these instances (repetitions) can be computed according to the task period and the system hyper-period ($\frac{hp}{T_i}$). We distinguish two types of multi-periodic systems:

Harmonic periodic system

A harmonic periodic system is composed of a set of harmonic and periodic tasks. Each task is executed according to a precise activation period which is an integral multiple of all lower periods. Tasks with harmonic periods are widely used to model real-time applications. They are easy to handle using some specific designs and scheduling algorithms.

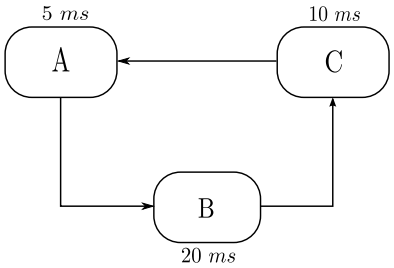


Fig. 2.4 Example of a harmonic multi-periodic system. The tasks periods are equal $T_A = 5\ ms$, $T_B = 20\ ms$ and $T_C = 10\ ms$.

Figure 2.4 illustrates an example of harmonic periodic system. This latter is composed of three periodic communicating tasks which are respectively executed at 5, 10, and 20 ms . The system hyper-period is equal to 20 ms.

Non-harmonic periodic system

A non-harmonic system is constituted by a set of periodic non-harmonic tasks. Tasks periods are chosen to match the application requirements (physical phenomena) rather than the implementation requirements (frequencies required by hardware or implementation details). These systems are not particularly amenable to a cyclic executive design. In addition, they have low scheduling theoretic.

This thesis proposes a general model which deals with harmonic and non-harmonic systems. In the following section, we formulate the first two problems studied in the thesis.

2.2 Problems 1 & 2: Modeling multi-periodic systems and evaluating their latencies

Modeling communications within multi-periodic systems, such as control/command applications, is a complex process. These systems are highly critical. The fundamental

requirement is to ensure the system functionality with respect to the data exchange between its parts as well as their temporal constraints. These systems require a deterministic behaviour, which means that for a given input the system execution must produce the same output. In addition, the system's execution must be temporally deterministic as well, always having the same temporal behaviour and respecting several hard real-time constraints.

Data flow models are a class of formalisms used to describe in a simple and compact way the communications of regular applications (for example video encoders). A model of this class is usually represented in the form of a network of communicating tasks where their executions are guided by the data dependencies. In this context, the first problem addressed in this thesis is:

How can we model the communications within multi-periodic systems, while meeting simultaneously their temporal and data requirements?

The second problem of this thesis is:

How can we evaluate the latency of a multi-periodic system?

We detail in chapter 4 the state of the art of modeling multi-periodic systems and evaluating their latencies using data flow formalisms. Next section lists several research approaches of modeling real-time systems.

2.3 Modeling real-time systems

Real-time systems are continuously interacting with external environment (other systems or hardware components). Assurance of a global quality of this kind of interactions is a major issue in the systems design. Nowadays, it is widely accepted that modelling plays a central role in the systems engineering. Modeling real-time systems provides an operational representation of the system overall organization as well as the behaviour of all its sub-systems. Therefore, using models can profitably substitutes experimentation. Modeling advantages can be summarized as follows [107]:

- flexibility : the model and its parameters can be modified in a simple way;
- generality by using abstraction: solving the scale factor issue;

- expressibility: improving observation, control and avoidance perturbations due to experimentation;
- predictability: real-time system characteristics can be predictable such as the system latency;
- reduced cost: systems modeling is less expensive than real implementation in terms of time and effort.

2.3.1 Models for Real-Time Programming

In general, the time representations in a real-time system can be classified into three types [74]: asynchronous, timed and synchronous models. These time models differ mainly in the manner of characterizing the correlated behaviour of a real-time system. More precisely, the time models define the relationship between the physical-time and logical-time which depends on several factors, such as the execution platform performance and utilization, the scheduling policies, the communication protocols as well as the program and compiler optimization.

Asynchronous model

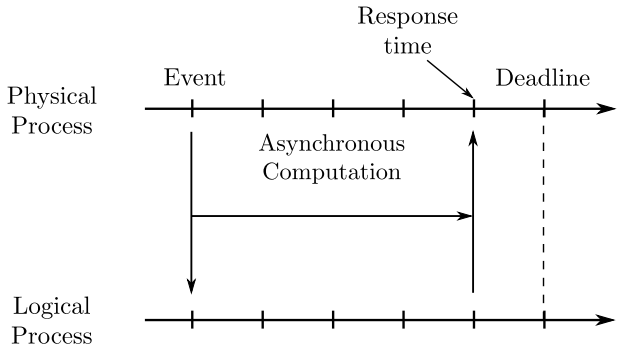


Fig. 2.5 Execution of an asynchronous model. Logical-time $\leq$ Physical-time.

Asynchronous model is a classical representation of real-time programming [118] such as the real-time programming languages "Ada" or "C". Applications are represented by a finite number of processes (tasks). Logical-time in the asynchronous model corresponds to the task's processing time (see Figure 2.5). The system execution is controlled by a scheduler setting the execution dates of the application's tasks. A task's execution depends on the adopted scheduling policy within the Real-Time Operating

System (RTOS in short). Accordingly, in asynchronous model the logical-time is not a priori determined and may vary depending on several factor such as the platform performance or the scheduling scheme. Therefore, asynchronous model logical-time variation is constrained by real-time deadlines: logical-time must be less than or equal to physical-time. The system schedulability condition (i.e. all deadlines are met) must be indicated in the scheduling scheme of real-time operating system. A schedulability analysis also requires the analysis of the worst-case execution times.

Timed model

The principle of timed model is based on the idea that interactions between two processes can only occur at two precise moments τ and $\tau^{'}$. The interval between these moments $(]\tau, \tau^{'}[)$ is a fixed non-zero amount of time which corresponds to the logical duration of computation and communication between two processes (see Figure 2.6).

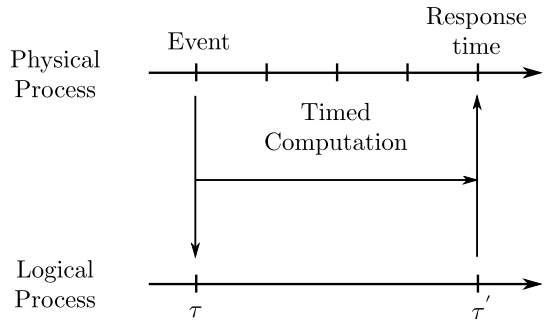


Fig. 2.6 Execution of a timed model. Logical-time = Physical-time.

In this context, programmers annotate their programs describing real-time systems with logical durations. The latter are an approximation of the actual computation and communication times. In this case, logical-time and physical-time are equal. Unlike the asynchronous model, since all the necessary information are known before execution, verification of some properties related to the logical temporal behaviour of the system becomes possible. For example, a scheduling validity or the system latency evaluation can be checked using a static method at compile-time.

In the timed model the program's logical execution time is fixed, two situations may arise at the execution level:

- The program may not have finished executing while its logical execution time is exceeded. If the timed program is indeed late, a runtime exception may be thrown (an exception is usually generated).

- The program may have completed its execution before its logical delay expiration. If the timed program finishes early, its output is delayed until the specified logical execution time has elapsed. The model annotation values should be carefully chosen in order to reduce the phase shift between the computations and output productions.

The timed model is well-suited for embedded control systems, which require timing predictability. The timed programming language Giotto [61] supports the design and implementation of embedded control systems [75], in particular, on distributed hardware. The key element in Giotto is that timed tasks are periodic and deterministic with respect to their inputs and states. The logical execution time of a Giotto task is the period of the task. For example, a 10Hz Giotto task runs logically for 100ms before its output becomes available and its input is updated for the next invocation.

Synchronous model

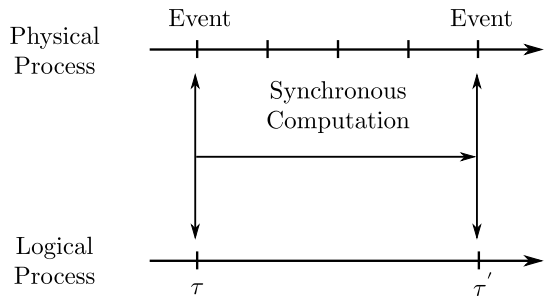


Fig. 2.7 Execution of a synchronous model. Logical-time = 0.

The principle of the synchronous model is similar to that of the timed model, in the sense that the two processes do not communicate between instants τ and τ'. In contrast to the timed model, the synchronous programmer assumes that any computational activity of a synchronous program including communication takes no time: logical-time in a synchronous program is always zero (see Figure 2.7). In other words, the synchrony hypothesis is satisfied as long as the system is fast enough to react to inputs coming from external environment and produces corresponding outputs before the acquisition of the next inputs. The interval $]\tau, \tau'[$ is seen as part of the logical-time model. In order to have a detailed the system behaviour, this interval can be refined by considering intermediate instants from the logical time point of view.

In order to model the communications within multi-periodic systems, we characterized the systems behavior using a temporal approach (a timed model). We assume that all the tasks parameters are provided by the developer.

2.3.2 Low-level languages

A large real-time systems class is implemented using low-level languages. However, the use of these languages presents several major disadvantages:

- Time-triggered communications are hard to implement. In fact, a part of the system should be scheduled manually.
- A basic mechanism for tasks communications is the "rendez-vous", which is implemented by three primitive operations: send, receive and reply. This type of communications may lead to deadlocks such as two communicating tasks are waiting each other to resume their executions.
- A low level of abstraction makes the program correctness harder to verify (manually or using automated tools).
- The use of low-level languages makes the program maintenance difficult. This is due to lack of informations differentiating parts of the program corresponding to real design constraints and parts which only correspond to implementation concerns.

Dealing with such considerations cannot be avoided. Actually, real-time systems developers tend to use programming languages with a higher-level of abstraction, where these issues are managed by the language compiler. Several languages seek to cover the entire process (from verification up to implementation). Their approach is based on translating the input program modeled with a high-level language into a low level code using automatic code generation processes. Such a strategy reduces the required time for implementation. Then, it provides a simple system description which is easier to analyse using verification tools. Finally, the compilation ensures the low-level code correctness.

2.3.3 Synchronous languages

The synchronous approach presents a high level of abstraction. This approach is based on simple and solid mathematical theories, which ease the program implementation

and verification. Synchronous approach introduces a discrete and logical time concept defined as succession of *steps* (*instants*). Each step corresponds to the system reaction. The synchrony assumption consider that the system is fast enough to react to its environment stimulus. This practically means that the system changes occurring during a step are treated at the next step and implies that responsiveness and precision are limited by the step duration.

The synchronous execution model has been developed in order to control in deterministic way reactions and interactions with the external environment. Esterel [16], Lustre [56] and Signal [13] are examples of synchronous languages. In order to meet the synchrony assumption, the correct implementation of these languages requires the estimation of the worst case execution times of the program code.

Lustre

Flow or *Signal* is an infinite sequence of values. It describes the values taken by the variables or expressions manipulated by a synchronous language. The sequence *clock* indicates the lack or the availability of a value carried by this sequence at a specific instants. The diverse features of Lustre programming language are illustrated in the following example:

```
node NOD (i: int; b: bool) returns (o: int)
var x: int; y: int when b;
let
        x = 0 -> pre(i);
        y = x  when b;
        o = current(y);
tel
```

A program consists of a set of equations that are hierarchically structured using nodes. These equations defines the program output flows according to the input flows. Equations use variables of either primitive types (int, bool,...) or user-defined types. They also use a set of primitive operators (arithmetic, relational ...). In addition to those operators, there exists four temporal operators:

- "**pre**"(previous) operator acts as memory. It allows to refer at cycle n to the value of a flow at cycle $n-1$. If **X**= $(x_1, x_2, \ldots, x_n, \ldots)$ is the values sequence of the flow **X**, then **pre(X)**= $(nil, x_1, x_2, \ldots, x_{n-1}, \ldots)$ In this flow, the first value is the undefined (non initialized). In addition, for any $n > 1$, its nth value of **pre(X)** is the $n-1$th value of **X**.

- "**->**"(follow by) operator is used to initialize a flow. If **X** $= (x_1, x_2, \ldots, x_n, \ldots)$ and **Y** $= (y_1, y_2, \ldots, y_n, \ldots)$ are two flows of the same type, then **X -> Y**$= (x_1, y_2, \ldots, y_n, \ldots)$. This means that **X -> Y** is equal to **Y** except the first value.
- "**when**" operator is used to create a slower clock according to a boolean flow such as **B** $= (true, false, \ldots, true, \ldots)$. **X when B** is the flow whose clock *tick* when **B** is *true* and its values are equal to those of **X** at these instants.
- "**current**" operator is used to get the current value which is computed at the last clock *tick* of the flow. If **Y = X when B** is a flow, then **current(Y)** is a flow having the same clock as **B** and whose value at a given clock's tick is the value taken by **X** at the last clock's tick when **B** was *true*.

i	1	2	3	4	5	...
b	*true*	*true*	*false*	*false*	*true*	...
pre(i)	*nil*	1	2	3	4	...
x = 0 **->** **pre** (i)	0	1	2	3	4	...
y = x **when** b	0	1			4	...
o = **current**(y)	0	1	1	1	4	...

Table 2.1 Lustre program behaviour.

The Lustre program behaviour is depicted in table 2.1. The flow value is given at clock's tick. The node output (o) is computed according to the inputs (i,b). x and y are two local (intermediate) variables. x = 0 **->** **pre** (i) equation sets x to 0 initially, and the subsequent x is equal to the value of i at the previous clock's tick. y = x **when** b means that y is present only when b is *true* and it takes the value of x at this instant. Finally, the output o is defined as the current value of y when it was available.

Due to the complexity of high-performance applications and to the intrinsic combinatorics of synchronous execution, multiple clock domains have to be considered at the application level. This is the case of a single system with multiple input/output associated with several real-time clocks. In this context, we focussing our study on modelling multi-periodic systems. In the following part, we introduce two synchronous languages dedicated to model multi-periodic systems.

Lucy-n

Synchronous languages such as Lustre [56] or Lucid Synchrone [103] define a restricted class of Kahn Process Networks which can be executed without buffers. For some appli-

cations (real-time streaming applications), synchrony condition forces the programmer to implement manually complex buffers, which is very error-prone. In order to avoid this issue, Cohen et al. [27] generalized the synchronous approach by introducing the n-synchronous approach. This latter is based on defining a relaxed clock-equivalence. Communication between n-synchronous streams is implemented with a buffer of size n. Accordingly, quantitative information about clocks can be revealed so that the compiler can decide whether it is possible to buffer a stream into another or not.

Based on the n-synchronous approach, Mandel et al. [89] introduced Lucy-n an extension of Lustre with a build-in buffer operator. The purpose of this language is to relax synchrony constraints while ensuring determinism and execution in bounded time and memory. Lucy-n handles the communication between processes of different rates through the buffers. A clock analysis is used to determine where finite buffers must be introduced into the data flow graph. Finally, clocks and buffer sizes are computed by the compiler.

In this thesis, tasks parameters are extracted from the real-time application itself. In addition, communication between two periodic tasks are modeled using an non-flow-preserving semantics.

Prelude

Prelude [42, 41] is a real-time programming language inspired by Lustre [56]. It focuses on the real-time aspects of multi-periodic systems. Predule is an integration language that assembles local mono-periodic programs into a globally multi-periodic system.

A program consists of a set of equations, structured into nodes. Real-time constraints representing the environment requirements are specified either on nodes' inputs/outputs (e.g. periods, deadlines) or on nodes (worst-case execution time). Equations of a node define its output flows according to its input flows. In order to manage the communication between nodes with different rates, transition operators are added to the synchronous data flow model. These operators are formally defined using strictly periodic clocks. They accelerate, decelerate, or offset flows. Therefore, they allow the definition of communication patterns provided by the user. The transition operators in Prelude allow the oversampling data sent from lower frequency tasks and under-sampling data sent from higher frequency tasks. Consequently, communications between nodes are non-flow-preserving (quasi-synchrony approach [23]).

The compiler translates a Prelude program into a set of communicating periodic tasks that respect the semantics of the original program. The tasks set is implemented as concurrent "C" threads that can be executed on a standard real-time operating

system. In case of a mono-processor platform [43], Prelude compiler implements deterministic communications between tasks without synchronization mechanisms. In order to establish these communications, the compiler ensures that the producer write before the consumer reads. In addition, it prevents a new execution of the producer to overwrite the value of its previous execution, if this latter is still required by other executions. Tasks parameters (release dates and deadlines) were adjusted in order to be scheduled on a mono-processor using Earliest Deadline First policy [85].

Our approach differs from [42, 41] in the following points: we characterize the systems behavior using a temporal approach (a timed model), in order to model the communications of multi-periodic systems. In addition, we tackle the mono-processor scheduling problem for applications modeled as non-preemptive strictly periodic tasks. Unfortunately, it has been proven [45] that the schedulability conditions for preemptive scheduling become, at best, necessary conditions for the non-preemptive case.

2.3.4 Oasis

Based on Time-Triggered Approach [77], Oasis [24, 87] is a framework dedicated to model and implement safety-critical real-time systems. An Oasis application is defined as a set of parallel communicating tasks called agents. Each agent is an autonomous running entity composed of a finite number of basic processing operations.

An Oasis application is implemented using an extension of "C" programming language, denoted by "ΨC". This language allows the tasks specification as well as their temporal requirements and interfaces. Each task t (agent) is characterized by a real-time clock H symbolizing the physical moments at which the (input/output) data can be observed. Clocks tasks are computed according to a global (smallest) clock which include all the observable moments of the system. Each basic processing is associated with a time interval defining its earliest start time (release date) and its latest end time (deadline). This latter is also the release date of the next processing. Such specific temporal dates are called temporal synchronization points.

There are two communication mechanisms in the Oasis model. The first one is shared variables (temporal variables) defined as real-time data flows. Variables can be shared implicitly between tasks and their past values of can be read by any task that needs it. Temporal variables modifications are only made visible at synchronization points. The second mechanism for data communication is explicit message passing. Each message defines a visibility date specified by the emitting task. In fact, the message can be observed by the receiving task from this date. Moreover, messages are stored, according to their visibility, in queues that belong to the receiving task.

Our communication model is similar the temporal variable mechanism in the sense that data is available at the emitting task deadline and can only consumed (received) at release date of the receiving task. However, in our model the release dates and the deadlines of successive executions do not necessary coincide. In addition, communications between the tasks executions is modeled, in a deterministic way, using a Synchronous Data Flow Graph whose size is equal to the application size.

2.3.5 Matlab/Simulink

Simulink [67] is a high-level modelling language developed by Mathworks. It is a specification and simulation tool widely used in the industrial field. Simulink models an application as a set of functional blocks (or Synchronous Block Diagrams) connected by signals. Each block is characterized by a set of input/output signals and an internal state. Each block output depends on its inputs and/or on its state. However, the block state depends only on the inputs of the block. A Simulink block transforms its inputs to outputs at a rate corresponding to its sample time. The block sample time indicates when the block will be executed during simulation. In other word, it indicates when the block internal state is updated and its outputs are computed. This sample time can be periodic or continuous. A periodic block is activated at time steps multiple of its period. Multi-periodic systems are modelled by a Simulink block diagram whose links defined the data transfer mechanisms between blocks.

In [76], we introduce an approach that models formally the synchronous semantic of multi-periodic Simulink systems using Synchronous Data Flow Graph. We proved that communication links between two periodic Simulink blocks can be modeled by a Synchronous Data Flow graph buffer. Our approach is based on a formal equivalence between the data dependencies imposed by the communication mechanisms in Simulink and the precedence constraints of a Synchronous Data Flow Graph. This approach allows multi/many-core implementation analysis since the resulting graph has the same size as the original Simulink system.

2.3.6 Architecture Description Language

An architecture description language (ADL) provides a high level of abstraction. The description may cover software features such as processes, threads and data as well as hardware component such as processors, memory, and input/output devices. This language focuses on describing the interaction between high level components of a system. The development process of a complex system, such as control/command

application, can be supported by such approach. The software engineering community uses ADLs in order to describe the software architectures. ADLs advantages are:

- Providing a formal representation of architectures.

- Both human and machines can apprehend their designs.

However, ADLs have a major drawback. There is no universal agreement on what ADLs should represent, especially when it is about the system behaviour.

In the sequel, we describe two languages dedicated to model multi-periodic systems.

Architecture Analysis and Design Language

Architecture Analysis and Design Language (AADL) [40] is used to model the software and hardware architecture of a real-time embedded system. An AADL design consists of a set of components interacting through interfaces. Components can be separated into three categories: application software (process, thread, etc.), execution platform (hardware) or composite components (composition of other components).

In AADL, user can specify a set of properties and extend them in order to fulfil his own requirement. For instance, a variety of execution properties can be assigned to threads, including timing (such as worst case execution times), dispatch protocols (such as aperiodic, periodic), memory size, etc.

Interactions among components are represented through connections defined between interface elements. Communication mechanisms can either be immediate (between simultaneous dispatches of the communicating threads) or delayed (from the current dispatch of the sender to the next dispatch of the receiver). Immediate communications impose a precedence order from the data sender to the receiver. However, in delayed communications, data are available to a receiving thread is that value produced at the most recent deadline of the emitting thread. Immediate and delayed communications can also be established between threads of different periods.

The AADL approach allows to model multi-periodic systems with a high-level of abstraction. The main restriction of this approach is that it provides a limited choice of communication schemes. However, our approach allows the definition of several (user-provided) communication schemes, while relying on the solid mathematical background of the SDFGs.

GIOTTO

Giotto [62] is a time-triggered Architecture Description Language designed to implement real-time systems. These latter can be executed in a distributed environment. A Giotto

program allows to specify the possible interactions systems' components and their environment.

A Giotto program is composed of four major concepts:

- Tasks: Giotto tasks are periodic. They are executed at regularly spaced points in time. They express calls to functions and they can have parameters defined by input and output ports.

- Ports: They allow the communication between a system and its environment. There are three types of ports: input, output and private. Private ports are used to communicate between tasks.

- Modes: They represent the major elements of a Giotto program. Giotto can only be in one mode at a certain instant. A mode consists of a period, an output set, actuators update, tasks execution and a set of switch modes.

- Drivers: They are functions that provide values for the input ports from sensors and mode ports. Drivers can be guarded: the guard of a driver is a predicate on sensor and mode ports. In other words, a task is executed only if the driver guard is *true*. Otherwise, the task execution is skipped.

A Giotto program consists of a set of modes. Each mode repeats the execution of a fixed set of tasks. These tasks are multi-periodic: each mode has a period P and each task has a frequency f, thus the task period is given by $\frac{P}{f}$.

Data produced by a task are available only at the end of its period. Conversely, when a task is executed it consumes only the last inputs' values, which is produced at the beginning of its period. This communication implies significant latencies between input data and the corresponding output one. However, our communication model considers that the data are available at the task deadline instead of the end of the task period ($D_i \leq T_i$).

2.4 Scheduling real-time system

Real-time scheduling problem consists in defining the tasks' execution order on processors of a given architecture. In real-time systems, tasks are subject to temporal constraints and possibly other constraints. The scheduling purpose is to set a start (preemption/recovery) date for each task execution in order to respect the system constraints.

A scheduler provides a policy for ordering the tasks executions on processors according to some predefined criteria. A scheduler is said to be optimal with respect to a given class of schedulers if and only if it generates a valid schedule for any feasible task set in this class [45]. In other words, a scheduling algorithm is said to be optimal regarding a system and a task model if it can schedule all tasks' sets that comply with the task model and are feasible on the system [33]. We distinguish several scheduling classes:

Offline and online scheduling

A scheduling is offline if all the decisions are made at compile-time (before the application execution). A table is generated containing the scheduling sequence that will be repeated infinitely during run-time. In order to be efficient, this approach requires a priori knowledge of all tasks parameters. Therefore, this approach is workable only if all the tasks are actually periodic. This approach is static and does not adapt to environment changes. However, offline scheduling is often the only way to predict whether temporal constraints are satisfied or not in a complex real-time system. As the scheduling is offline, the use of exact algorithms is possible even if these methods are very time consuming.

A scheduling is online if decisions are made during the runtime of the system. At each moment of the system life time, the scheduling algorithm is able to process tasks that have not been previously activated using the task parameters specific to this moment. This makes online scheduling flexible and able to adapt to environment changes. In this case, the use of effective heuristics is preferred to that of optimal algorithms.

Independent/dependent tasks Scheduling

The scheduling policy is divided into dependent and independent tasks scheduling. Tasks whose execution is not dependent upon other tasks' executions are termed independent tasks. These latter have no dependencies among the tasks and have no precedence order to be followed during scheduling. In this case, we talk about independent tasks scheduling.

Given a real-time system, dependency between tasks ensures that the tasks are executed in some order. Executing a dependent task may require the availability of data provided by other tasks of the system. Therefore, this task can only be schedule

after the compilation of its predecessors. The relationship between independent tasks is typically described by a directed graph called precedence or communication graph.

Preemptive and non-preemptive scheduling

A scheduling is called preemptive if the task's execution can be suspended and resumed later on without affecting its behaviour, except by increasing its elapse time. A task is typically suspended by an other task with higher priority, when this latter becomes ready to be executed. Additional cost caused by the preemption may not be negligible compared to the tasks execution and communication times. Moreover, the system may include critical sections where preemption is not allowed.

In contrast, a non-preemptive scheduler does not suspend tasks. When a task is selected to be executed, it runs until the end of its execution. In this case, a higher priority task cannot interrupt the execution of a lower priority task even when it becomes ready to be executed. Compared to preemptive scheduling, non-preemptive algorithms are easy to implement and have a low runtime overhead. The time consumption of each task in the system can be easily characterized and predicted [65]. Even though the non-preemptive scheduling has been proven to be NP-Hard in strong sense [80], this type of scheduling is widely used in industrial applications such as cell phone devices [100] and mobile robotics [101].

Mono-processor and Multi-processor scheduling

Designing real-time systems is influenced by the hardware architecture on which the application may run. It designates the physical resources needed to control the process. This architecture is composed of processors, input / output cards, memories, networks, etc. According to the processors number, we distinguish two scheduling problems:

- Mono-processor scheduling: All the system tasks are executed by a single processor.
- Multi-processor scheduling: In order to be executed, the application's tasks are assigned to n processors sharing a central memory.

In the following section, we formulate the scheduling problem studied in the thesis.

2.5 Problem 3: Scheduling strictly multi-periodic system with communication constraints

Nowadays, real-time embedded systems are becoming increasingly complex. Consequently, these systems require additionally Real-Time Operating Systems (RTOS in short) to manage concurrent tasks and critical resources. Most RTOS implement real-time dynamic scheduling strategies such as Rate Monotonic, Earliest Deadline First and so on. The real-time applications are modeled as a set of periodic tasks. Each task is characterized by a set of parameters such as period, release date, deadline and so on. Several schedulability conditions have been provided by the real-time community. These conditions verify, prior to the implementation phase, whether a set of tasks will meet their deadlines for a given scheduling strategy and platform.

Several studies addressed the preemptive scheduling problem of periodic tasks set. However, much less attention was given to the non-preemptive one. In fact, preemptive scheduling has a good task responsiveness, however it increases the resource utilization rate compared to non-preemptive scheduling. Nonetheless, there exists several cases where non-preemptive scheduling is preferable. For instance, the overhead generated by preemption is not always negligible compared to task execution time and inter-processor communication. Non-preemptive scheduling on a uniprocessor guarantees exclusive access to shared resources and data as well as eliminating the need for synchronization and associated overhead. On the other hand, many industrial applications (e.g. avionic applications [1]) are modeled as a set of strictly periodic tasks. This strict periodicity is usually essential to avoid problems arising form input and output jitters. For example, input jitter can cause a data loss from sensors. Output jitter may lead to performance degradation of the systems, as its negligence can be a source of instability.

On the other hand, communication within multi-periodic systems is more and more indispensable. Due to the tasks strict deadlines that must be met, communications between the tasks executions are implemented in a completely deterministic manner. In this context, the third problem we address in this thesis is:

How to schedule on mono-processor a set of strictly periodic tasks, assuming that these tasks communicate with each other and can not be preempted?

We detail in chapter 4 the state of the art regarding this scheduling problem.

2.6 Conclusion

In this chapter, we introduced the context of this thesis. We presented an overview of real-time systems and their characteristics. We have formulated the two first problems that we studied in this thesis: modeling communications within multi-periodic systems and evaluating their latencies. In addition, we have reviewed several existing approaches regarding multi-periodic systems design. The novelty of our approach consists in using Synchronous Data Flow Graphs for modeling communications within multi-periodic systems in a deterministic way. Afterwards, we have introduced the basic notions of real-time scheduling. Finally, we formulated the third problem studied in this thesis: mono-processor scheduling of non-preemtive strictly periodic set of tasks with communication constraints.

The following chapter introduces an overview of data flow models. Moreover, it introduces notions and transformations related to the Synchronous Data Flow Graphs.

Chapter 3

Data Flow models

The emergence of microelectronics and modern calculators led to the development of increasingly accurate models. These models seek to formally describe the notion of parallel and distributed computing. Kahn Process Network is one of the first attempts which models an application using First In First Out queues between the application sub-parts. However, this model is too permissive to allow an algorithmic prediction of the application performances. Nowadays, Synchronous Data Flow graph is a widely used model which provides a compromise between the application expressiveness and analyse.

This chapter provides an overview of data flow models and introduces important notion concerning them. Section 3.1 presents an overview of data flow models. Section 3.2 introduces notions and transformations related to Synchronous Data Flow Graph model. Section 3.3 concludes the chapter.

3.1 Data Flow Graphs

The data flow model is inspired from computation graphs and Kahn Process Networks (KPN). This model is based on a graphical representation of programs and architectures. In the model's original definition [34, 32], the graph vertices represent the elementary operations of the computer program and the graph edges between two vertices model the data dependencies between two operations. The program's initial conditions are modeled by adding tokens on the graph edges. In Denis' model [34], an operation which is represented by a graph vertex, can be executed whenever data are available on all its inputs. When an operation is executed, it consumes a token on each input edge and produces a token on each output edge. The data flow representation is dedicated to computer programs, multi-processors architecture and parallel compilation.

In the sequel, we present data flow models that are frequently used for embedded systems design. These models are: Kahn Process Network, Computation Graph, Synchronous Data Flow Graph and Cyclo-Static Data Flow Graph.

3.1.1 Kahn Process Network

We start by presenting Kahn Process Network (KPN in short) [46] since it is one of the most general data flow model. KPN is a data flow model based on a graphical description. It describes an application using a directed graph. the vertices symbolize the application sub-sections (processes). During the application execution, processes communicate through communication channels. These channels are unbounded FIFO (First In First Out) buffers which temporarily store data. The input process can exclusively write into the buffer and the output process can exclusively read from the buffer. Reading from an empty buffer is blocking. In fact, when a process starts reading from a buffer, its execution is suspended until the data arrival (it must wait for the data availability before resuming its execution). The process cannot verify the presence of data before its execution. In addition, the processes are not reentrant. This means when a process starts executing, it must finish before starting an other execution.

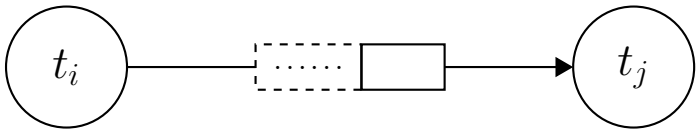


Fig. 3.1 Example of Kahn Processor Network.

The Kahn Process Network is a deterministic model. Indeed, the output data of the KNP depend only on the input data. This output value is neither influenced by the processes execution times and their executions order nor by the communication delays between them. Moreover, the semantic of this model does not allow the buffer sizing. In order to compensate this issue, the buffer has infinite size. Thus, this model is useful to describe complex applications. However, it is unable to predict the system performances.

3.1.2 Computation Graph

Computation graphs [69] (CG in short) are one of the first static data flow model. Karp and Miller defined the CG in order to propose an alternative to sequential execution

model. As the Kahn Process Network, an application (system) is represented by directed graph. The graph nodes model the tasks (or sub-tasks) of an application. The graph arcs represent the data exchange between these tasks. Each arc corresponds to an unbounded FIFO (First In First Out) buffer connecting an input task to an output one. At the end of the input task execution, a fixed amount of data is produced and stored temporarily in the buffer. According to a fixed threshold, these data are subsequently consumed from the buffer at the beginning of the output task execution. Each task activation depends only on the availability of a sufficient amount of data on all of its input arcs. In addition, each buffer may contain an initial amount of data.

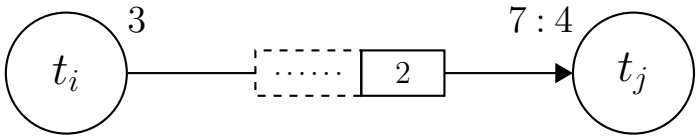


Fig. 3.2 Example of Computation Graph.

In the sequel, we referred data as tokens. Figure 3.2 represents a Computation Graph consisting of two tasks t_i and t_j connected by a buffer $a = (t_i, t_j)$. The initial amount of tokens is equal to 2. Each execution of t_i produces 3 tokens in the buffer a. In order to be able to execute, t_j must wait for the availability of 7 tokens in the buffer, since the required data threshold is equal to 7. At the beginning of its execution, t_j consumes 4 tokens from the buffer a. Note that the data threshold must be greater or equal than the amount of tokens consumed by t_j.

Some applications are represented using the Computations graphs model which highlights the parallelism between their different parts. Karp and Miller [69] show that the execution of a program modeled by a CG is deterministic (static). It is possible to study the program behaviour even without its execution. Accordingly, several fundamental problems become decidable such as checking the application liveliness (absence of deadlock).

3.1.3 Synchronous Data Flow Graph

The "Synchronous Data Flow Graphs" (SDFG in short) are a formalism introduced and studied by Lee and Messerchmidtt [83, 84] in order to model the communications within an embedded applications. They are defined by a set of tasks exchanging data using FIFO queues. A SDFG is represented by an oriented graph $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. SDFGs can be considered as a particular case of Computation Graphs where the data thresholds of the buffers are equal to the consumption rates.

A Synchronous Data Flow Graph is defined as follows:

- Each node $t_i \in T$ corresponds to a task. Nodes can be executed infinity of times.
- Each arc $a = (t_i, t_j)$ corresponds to a buffer formed by an unbounded FIFO queue connecting an input task t_i to an output task t_j.
- A buffer may contain data which are referred as tokens. The initial amount of tokens in a buffer is expressed by $M_0(a)$. This notation is also called initial marking.

At the beginning of their executions, tasks consume tokens from their inputs. Furthermore, these tasks produce tokens in their outputs at the end of their executions.

- in_a is the production rate which corresponds to the amount of tokens produced in the buffer a at each execution of the input task t_i.
- out_a is the consumption rate which corresponds to the amount of tokens consumed from the buffer a at each execution of the output task t_j.

An application modeled by a consistent SDFG can be always implemented. Provided that SDFG' tasks take finite time and finite memory in order to execute, the implementation will also take finite time and finite memory. Thus, an application modeled with a SDFG can be executed (infinitely) in a periodic way without requiring additional resources. This formalism suits well the streaming applications such as video encoders and decoders such as the H263 in [112]. In addition, SDFG formalisms can be used in some academics tools [115, 7, 51] or industrial ones [52] in order to map an application on a many-core architecture.

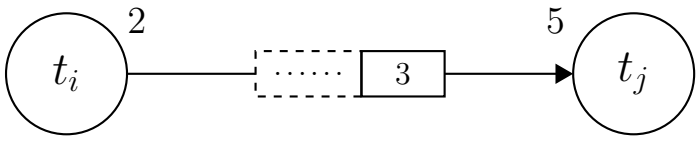


Fig. 3.3 A simple example of Synchronous Data Flow Graph.

Figure 3.3 illustrates a buffer $a = (t_i, t_j)$. The number of initial tokens in the buffer is equal to 3. At the end of each execution of t_i, 2 tokens are produced and stored temporarily in the buffer a. Task t_j cannot be executed unless the number of tokens in the buffer is at least equal to 5. Then, 5 tokens will be consumed at the beginning of the t_j execution .

Homogeneous Synchronous Data Flow Graph

There exists a particular case of Synchronous Data Flow Graph named Homogeneous Synchronous Data Flow Graphs (HSDFG in short). A Homogeneous SDFG is a SDFG where the production and consumption rates are equal $in_a = out_a$. By convention, we note it by

$$\forall a \in \mathcal{A},\ in_a = out_a = 1.$$

3.1.4 Cyclo-Static Data Flow Graph

"Cyclo-Static Data Flow Graph" (CSDFG in short) [17] is an extension of SDFG. In this model, the production and consumption rates are decomposed into phases. The amount of tokens consumed and produced by a task varies from one execution to the successive one in a cyclic pattern. These rates are represented as a vector.

Example 3.1.1 Consider a buffer $a = (t_i, t_j)$. Denote by $out_a(k_j)$ the quantity of tokens consumed during the k_jth execution of the task t_j with $k_j \in \{1, \ldots, \phi(t_j)\}$. Similarly, denote by $in_a(k_i)$ the quantity of tokens produced during the k_ith execution of the task t_i with $k_i \in \{1, \ldots, \phi(t_i)\}$.

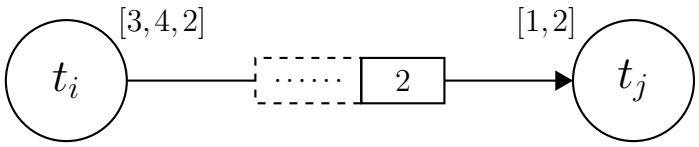


Fig. 3.4 An example of a Cyclo-Static Data Flow Graph.

As depicted in Figure 3.4, the task t_i will produce 3 tokens at the end of its first execution. Then, it produces 4 tokens after its second execution. Finally, it produces 2 tokens after its third execution. This sequence of $\phi(t_i) = 3$ executions is called an iteration. Once this iteration is over, we restart from the beginning.

Denote by i_a the total amount of data produced in the buffer a after an iteration of its input task and denote by o_a the total amount of data consumed from the buffer a after an iteration of its output task.

$$i_a = \sum_{k=1}^{\phi(t_i)} in_a(k) \text{ and } o_a = \sum_{k=1}^{\phi(t_j)} out_a(k)$$

3.2 Synchronous Data Flow Graph Characteristics and transformations

Several tests and transformations can be used to study the Synchronous Data Flow model. These transformations have different complexities. In the sequel, we present some essential transformations to our study such as expansion and repetition vector computation.

3.2.1 Precedence relations

Consider the SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$, for each task $t_i \in \mathcal{T}$, we denote by $t_i^{\nu_i}$ the ν_ith execution of t_i where ν_i is a strictly positive integer.

In the SDFG formalism, the task execution is conditioned only by the availability of sufficient amount of tokens on its inputs. For a buffer $a = (t_i, t_j) \in A$ with $M_0(a)$ as initial marking, the executions of t_j may require the presence of data produced by t_i. In this case, the buffer a induces a set of precedence constraints between the executions of t_i and t_j. We say that there exists a precedence relation between the ν_ith execution of t_j and the ν_jth execution of t_i if and only if:

- Condition 1: $t_j^{\nu_j}$ can be executed after $t_i^{\nu_i}$.
- Condition 2: $t_j^{\nu_j}$ can not be executed before $t_i^{\nu_i}$.

After the ν_ith execution of t_i, $\nu_i \cdot in_a$ tokens are added to the buffer a. On the other hand, at the beginning of the ν_jth execution of t_j, $\nu_j \cdot out_a$ tokens are consumed from the buffer. The resulting marking of these executions is equal to:

$$M(a) = M_0(a) + \nu_i \cdot in_a - \nu_j \cdot out_a \tag{3.1}$$

In the general case, Munier [96] defines a strict precedence relationship between the executions of adjacent tasks. Consider a SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Let $a = (t_i, t_j)$ be a buffer with $M_0(a)$ as initial marking. Let (ν_i, ν_j) be a pair of strictly positive integers. We say that there exists a precedence relation between $t_i^{\nu_i}$ and $t_j^{\nu_j}$ if the following conditions are met:

- Condition 1: $t_j^{\nu_j}$ can be executed after $t_i^{\nu_i}$. This condition is equivalent to:

$$M_0(a) + \nu_i \cdot in_a - \nu_j \cdot out_a \geq 0.$$

- Condition 2: $t_j^{\nu_j-1}$ can be executed before $t_i^{\nu_i}$, while $t_j^{\nu_j}$ cannot be executed before $t_i^{\nu_i}$. This condition is equivalent to:

$$in_a > M_0(a) + (\nu_i - 1) \cdot in_a - (\nu_j - 1) \cdot out_a \geq 0.$$

By combining these two inequalities, Munier [96] obtained the following theorem:

Theorem 3.2.1 *Let $a = (t_i, t_j)$ be a buffer with $M_0(a)$ as initial marking. Let (ν_i, ν_j) be a couple of strictly positive integers. There is a precedence constraint between the ν_ith execution of t_i and the ν_jth execution of t_j if and only if:*

$$in_a > M_0(a) + \nu_i \cdot in_a - \nu_j \cdot out_a \geq \max\{0, in_a - out_a\}. \tag{3.2}$$

In the remainder of this manuscript, we use the term "precedence constraint" to refer to a strict precedence constraint.

Example 3.2.1 Consider the SDFG of Figure 3.3. There is a precedence constraint between the executions t_i^4 and t_j^2 induced by the buffer a. Indeed, after the execution of t_i^4, $3 + 4 \times 2 = 11$ tokens are produced in the buffer a. Then, after the execution of t_j^2, $2 \times 5 = 10$ tokens are consumed from the buffer. Therefore, t_j^2 can be executed after t_i^4 (Condition 1).
After the execution of t_i^3, $3 + 3 \times 2 = 8$ tokens are produced in the buffer a. However, in order to execute t_j^1, $1 \times 5 = 5$ tokens are required. Therefore, t_j^1 can be executed before t_i^4, while, t_j^2 cannot be executed before t_i^4 since there are not enough tokens in the buffer (Condition 2).

Relation between the tasks executions' indexes

Consider a SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Let $a = (t_i, t_j)$ be a buffer with $M_0(a)$ as its initial marking. Let (ν_i, ν_j) be a pair of strictly positive integers such that there exists a precedence constraints between the ν_ith execution of t_i and the ν_jth execution of t_j. Marchetti and Munier [90] establish a relation between the execution indexes of two adjacent tasks. The following lemma describes this relation in detail.

Lemma 3.2.1 *Let $a = (t_i, t_j)$ be a buffer of a SDFG. Let $k_{min} = \frac{\max\{0, in_a - out_a\} - M_0(a)}{gcd_a}$ and $k_{max} = \frac{in_a - M_0(a)}{gcd_a} - 1$ be two integer values, with $gcd_a = gcd(in_a, out_a)$.*

1. *If buffer* (a) *induces a precedence constraint between the executions* $t_i^{\nu_i}$ *and* $t_j^{\nu_j}$ *then there exists* $k \in \{k_{min}, \ldots, k_{max}\}$ *such that*

$$in_a \cdot \nu_i - out_a \cdot \nu_j = k \cdot gcd_a.$$

2. *Conversely, for any* $k \in \{k_{min}, \ldots, k_{max}\}$*, there exists an infinite number of couples* $(\nu_i, \nu_j) \in \mathbb{N}^{*^2}$ *such that* $in_a \cdot \nu_i - out_a \cdot \nu_j = k \cdot gcd_a$ *and the buffer* (a) *induces a precedence relationship between the executions* $t_i^{\nu_i}$ *and* $t_j^{\nu_j}$

Considering the SDFG example in Figure 3.3, Buffer (a) and its initial marking induce a precedence relationship between the fourth execution of t_i and the second execution of t_j. According to the lemma 3.2.1, there exists an integer value $k \in \{k_{min} \ldots k_{max}\}$ where $k_{min} = \frac{\max\{0,2-5\}-2}{1} = -2$ and $k_{max} = \frac{2-2}{1} - 1 = -1$ with $gcd_a = \gcd(2,5) = 1$. Therefore, the relationship between the execution indexes of t_i^4 and t_j^2 is establish as follows:

$$in_a \cdot \nu_i - out_a \cdot \nu_j = 2 \cdot 4 - 5 \cdot 2 = k \cdot 1,$$

with $k \in \{-2, -1\}$.

3.2.2 Consistency

The consistency of a SDFG is a necessary condition for the existence of a schedule with bounded memory [84]. During the system execution, consistency ensures that the amount of tokens consumed is equal to that produced in each buffer of the SDFG.

Definition 3.2.1 *Let* $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ *be a SDFG. For each task* $t_i \in \mathcal{T}$*, we define the set of input arcs* $\mathcal{A}^+(t_i)$ *and the set of output arcs* $\mathcal{A}^-(t_i)$ *as follow:*

$$\mathcal{A}^+(t_i) = \{a = (t_j, t_i) \in \mathcal{A}, t_j \in \mathcal{T}\} \text{ and } \mathcal{A}^-(t_i) = \{a = (t_i, t_j) \in \mathcal{A}, t_j \in \mathcal{T}\}$$

The topology matrix of a SDFG allows a synthetic description of a system. This matrix is analogous to the incidence matrix in graph theory. Let us consider a SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Its topology matrix $\Gamma_{\mathcal{G}}$, of size $|\mathcal{A}| \times |\mathcal{T}|$, is defined in the following way: $\forall (a, t_i) \in \mathcal{A} \times \mathcal{T}$,

$$\Gamma_{\mathcal{G}}(a, t_i) = \begin{cases} in_a & \text{if } a \in \mathcal{A}^-(t_i) \\ -out_a & \text{if } a \in \mathcal{A}^+(t_i) \\ 0 & \text{Otherwise} \end{cases}.$$

Each column of $\Gamma_{\mathcal{G}}$ corresponds to a node and each row corresponds to an arc. G is consistent only when the rank of $\Gamma_{\mathcal{G}}$ is equal to $|\mathcal{T}|-1$.

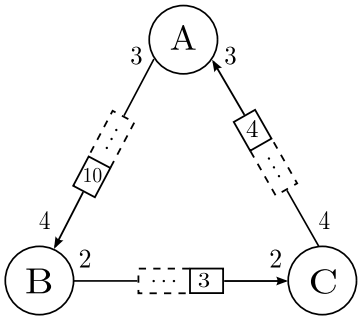


Fig. 3.5 Cyclic Synchronous Data Flow Graph.

Let us consider the SDFG depicted in Figure 3.5, its topological matrix is equal to:

$$\Gamma = \begin{pmatrix} 3 & -4 & 0 \\ 0 & 2 & -2 \\ -3 & 0 & 4 \end{pmatrix}$$

The SDFG consistency can be verified by computing the rank of the topology matrix which is equal to $|\mathcal{T}|-1=2$.

3.2.3 Repetition vector

An other way to check the SDFG consistency is to compute its repetition vector R [84]. The SDFG's repetition vector indicates for each task the minimal number of executions, such that the graph returns to its initial state. Hence, a SDFG is said to be consistent if its repetition vector exists.

We compute the SDFG repetition vector R by finding a vector with the smallest strictly positive integer entries which satisfies the following equation:

$$\Gamma_{\mathcal{G}} \cdot R^T = 0.$$

We note R_i the repetition factor of a task $t_i \in \mathcal{T}$. In order to compute the SDFG's repetition vector, each buffer should verify the following equality:

$$\forall a = (t_i, t_j) \in \mathcal{A},\ out_a \cdot R_i = in_a \cdot R_j.$$

Let's consider the previous SDFG represented in Figure 3.5. Its repetition vector can be computed using the following system:

$$\begin{aligned} 4 \cdot R_B &= 3 \cdot R_A \\ 3 \cdot R_A &= 4 \cdot R_C \\ 2 \cdot R_c &= 2 \cdot R_B \end{aligned}$$

By solving this system, we can deduce that the SDFG is consistent and its minimal repetition vector is equal to $R = [4,3,3]$. The graph reaches its initial state (initial marking), when A is executed 4 times, while B and C are respectively executed 3 times.

3.2.4 Normalized Synchronous Data Flow graph

The *normalization* [91] consists on transforming a SDFG into a normalized equivalent graph where all the arcs values adjacent to a task are equal. This transformation has no influence on precedence relationships; however it serves to simplify several stages of analysis. Marchetti and Munier [91] prove that any consistent Synchronous Data Flow Graph can be normalized in polynomial time.

Definition 3.2.2 *Let $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ be a SDFG. A task $t_i \in \mathcal{T}$ is normalized, if there is a strictly positive integer $z_i \in \mathbb{N}^*$ such that*

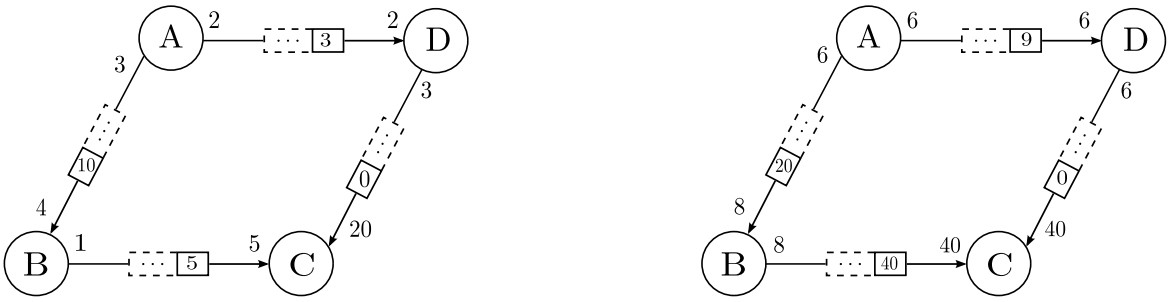
$$\begin{aligned} \forall a \in \mathcal{A}^-(t_i),\ in_a = z_i \\ \forall a \in \mathcal{A}^+(t_i),\ out_a = z_i. \end{aligned}$$

Definition 3.2.3 *A SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ is normalized if each task $t_i \in \mathcal{T}$ is normalized. Normalizing $\mathcal{G}$ consists on finding two strictly positive integer vectors $z = (z_1, \ldots, z_{|\mathcal{T}|})$ and $\mu = (\mu_1, \ldots, \mu_{|\mathcal{A}|})$ that verify the following equation system:*

$$\begin{aligned} \forall a \in \mathcal{A}^-(t_i),\ \mu_a \cdot in_a = z_i \\ \forall a \in \mathcal{A}^+(t_i),\ \mu_a \cdot out_a = z_i. \end{aligned}$$

In order to validate this transformation, the initial marking of each SDFG buffer must be multiplied by the corresponding factor as follows:

$$\forall a \in A,\ M_0(a) \cdot \mu_a$$



(a) Unnormalized SDFG. (b) Equivalent normalized SDFG.

Fig. 3.6 Example of Synchronous Data Flow Graph *normalisation*.

The normalization example shown in Figure 3.6b is obtained by solving the following equation system:

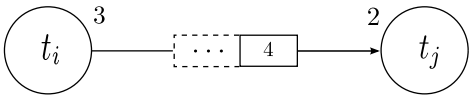
$$z_A = 3 \cdot \mu_{AB} = 2 \cdot \mu_{AD} \qquad z_B = 4 \cdot \mu_{AB} = 1 \cdot \mu_{BC}$$
$$z_c = 5 \cdot \mu_{BC} = 20 \cdot \mu_{DC} \qquad z_D = 2 \cdot \mu_{AD} = 3 \cdot \mu_{DC}$$

Since the SDFG of Figure 3.6a is consistent, we can deduce the existence of a solution before solving the equation system. A minimal solution of this system is equal to $\mu = (\mu_{AB}, \mu_{AD}, \mu_{BC}, \mu_{DC}) = (2,3,8,2)$ and $z = (6,8,40,6)$. Thus, the initial marking of each buffer is equal to $M_0(a_{AB}) = 20$, $M_0(a_{BC}) = 40$, $M_0(a_{DC}) = 0$ and $M_0(a_{AD}) = 9$.

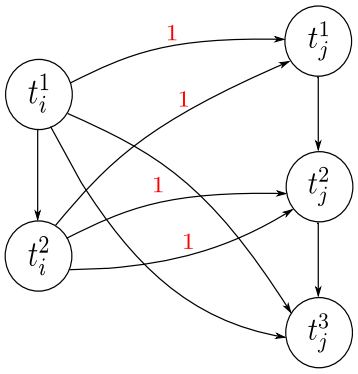
Normalization is an analytical tool that has been successfully used in the analysis SDFG liveliness conditions. In our study, we only deal with consistent and normalized SDFG.

3.2.5 Expansion

Synchronous Data Flow graphs may be analysed by transforming them into an equivalent Homogeneous Synchronous Data Flow Graph. HSDFGs are particular class of SDFGs where all the production and consumption rates are equal. Efficient analysis technique have been developed for this class of graphs. Most of these analysis which are applied to HSDFGs may be applied to the original graphs. For example, each valid schedule for the equivalent HSDFG is a valid schedule for the original SDFG. In contrast, the disadvantage of this transformation is that the size of the resulting graph (HSDFG) can increase, in the worst case, exponentially compared to the size of the SDFGs.



(a) An example of a SDFG.



(b) Equivalent Homogeneous SDFG.

Fig. 3.7 Example of a SDFG transformation into equivalent HSDFG using the technique introduced by Sih and Lee [108].

Based on the consistency [84], Sih and Lee [108] defined a SDFG transformation into an equivalent HSDFG. Their approach consists on duplicating the SDFG's tasks according to their repetition factors. In order to ensure the precedences relations, arcs are added according to the tasks production. In other words, the HSDFG arcs number corresponds to the number of tokens produced in the SDFG's buffers. Figure 3.7b illustrates the equivalent HSDFG resulting from the transformation of the SDFG depicted in Figure 3.7a. The equivalent HSDFG consists of 5 duplicates (tasks) connected by 9 buffers.

Munier [96] introduces the *expansion*, a transformation which associates to each SDFG buffer $a = (t_i, t_j)$ an equivalent HSDFG. Precedence constraints between the executions of t_i and t_j are fulfilled by adding $\min(R_i, R_j)$ arcs to the equivalent HSDFG, where R_i and R_j are respectively the repetition factor of t_i and t_j. The next part of this section explains in detail the different steps of this transformation.

Let $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ be a SDFG. Let $a = (t_i, t_j)$ be a buffer with $M_0(a)$ its initial marking and $R = (R_i, R_j)$ its repetition vector. Let (ν_i, ν_j) be a pair of strictly positive integer such that it exists a precedence constraint between $t_i^{\nu_i}$ and $t_j^{\nu_j}$. Hence, ν_i and ν_j satisfy the following equation

$$in_a > M_0(a) + \nu_i \cdot in_a - \nu_j \cdot out_a \geq \max\{0, in_a - out_a\}.$$

On the other hand, R_i and R_j are the tasks' repetition factors. This means that $R_i \cdot in_a = R_j \cdot out_a$. Hence,

$$in_a > M_0(a) + (\nu_i + R_i) \cdot in_a - (\nu_j + R_j) \cdot out_a \geq \max\{0, in_a - out_a\}.$$

This inequalities implies that there is an infinite set of precedence constraints which are repeated every iteration. An iteration means a phase in which tasks t_i and t_j are executed respectively R_i and R_j times.

The underlying idea of the *expansion*, is to represent the infinite set of precedence constraints induced by the buffer and its initial marking using a finite repetitive structure, i.e a pattern. This transformation is composed of three steps:

1. Each task t_i is replaced by R_i (task repetition factor) tasks denoted by $t_i^1, \ldots, t_i^{R_i}$. such that for any $k \in \{1, \ldots, R_i\}$ and $\alpha > 0$, the αth execution of t_i^k corresponds to the $((\alpha-1) \cdot R_i + k)$th execution of t_i. Tasks $t_i^1, \ldots, t_i^{R_i}$ are called duplicates of t_i.

2. For successive executions of each task, a buffer $a_k = (t_i^{k_i}, t_i^{k_i+1})$ is added for $k_i \in \{1, \ldots, R_i - 1\}$ with $M_0(a_k) = 0$. In addition, $a_{R_i} = (t_i^{R_i}, t_i^1)$ is added with $M_0(a_{R_i}) = 1$.

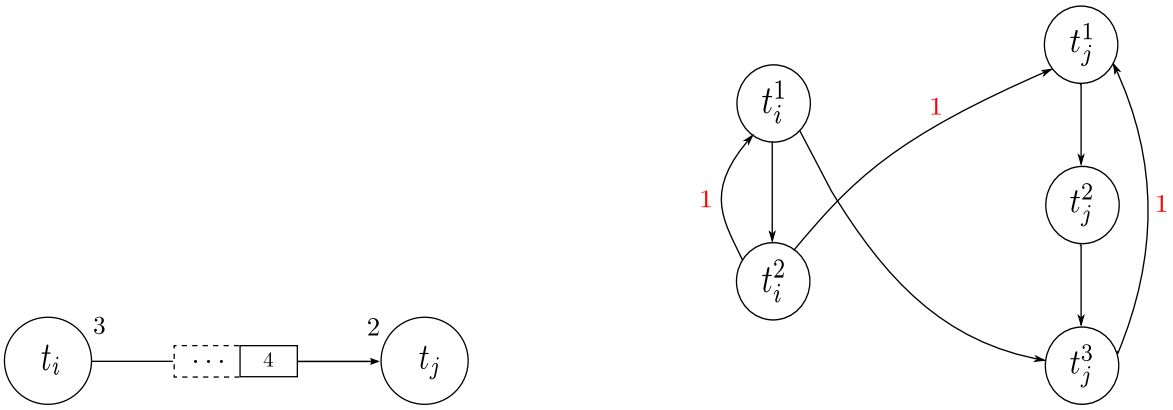
3. Arcs are added between $t_i^{\nu_i}$ and $t_j^{\nu_j}$ in the following cases:

   - If $in_a > out_a$, R_i arcs are added between the executions of t_i and t_j. The executions indexes are computed as follows:

   $$\forall \nu_i \in \{1, \ldots, R_i\},\ \nu_j = \lfloor \frac{M_0(a) + in_a \cdot (\nu_i - 1)}{out_a} \rfloor + 1$$

   - If $in_a \leq out_a$, R_j arcs are added between the executions of t_i and t_j. The executions indexes are computed as follows:

   $$\forall \nu_j \in \{1, \ldots, R_j\},\ \nu_i = \lceil \frac{out_a \cdot \nu_j - M_0(a)}{in_a} \rceil \text{ with } \nu_j > \frac{M_0(a)}{out_a}$$



(a) An example of a SDFG.

(b) Equivalent Homogeneous SDFG.

Fig. 3.8 Example of SDFG transformation into equivalent HSDFG using the *expansion*.

Figure 3.8b represents the equivalent HSDFG resulting from the *expansion* of the SDFG depicted in Figure 3.8a. The SDFG repetition factor is equal to $R = (2,3)$. We notice that the number of arc which models the precedence constraints between the executions of t_i and t_j is equal to $\min(2,3) = 2$.

Marchetti and Munier [90] have proven that the *expansion* of a SDFG is not polynomial. The size of the resulting HSDFG is pseudo-polynomial ($\mathcal{O}(|\mathcal{T}| \cdot \max_{t_i \in \mathcal{T}} R_i)$) according to the SDFG size. Hence, *expansion* is an efficient technique for instance with reasonable average repetition factor.

3.3 Conclusion

This chapter has introduced an overview of static data flow models. Important notions and transformations were introduced in the context of the SDFG model: precedence constraints, consistency, repetition vector, normalization and expansion. The sequel of this thesis addresses only a particular class of normalized SDFG. This particular class is used for the communication modeling problem of real-time applications, addressed in Chapter 5.

The next chapter introduces the data flow formalisms dedicated to model a real-time system and evaluate its latency. In addition, this chapter presents a state of the art on the scheduling problem of non-preemptive strictly periodic tasks.

Chapter 4

State of art

Modeling stage is essential for designing real-time systems. This procedure provides an operational representation of the system. Modeling an application using static data flow graph provides additional informations related to its execution. Thus, it is possible to evaluate accurately the system performances. On the other hand, these applications must be (properly) scheduled in order to be executed on a given platform. Schedulers must take into account several type of constraints such as resource and communication constraints.

This chapter gives an overview on the state of the art related to this thesis. The remainder of this chapter is organized as follows. Section 4.1 positions our study with respect to the state of the art for modeling real-time systems and evaluating their latencies using data flow formalisms. Section 4.2 compares our approach with some existing non-preemptive strictly periodic scheduling approaches. Conclusion is given in Section 4.3.

4.1 Modeling real time systems using data flow formalisms

Critical real-time applications are becoming increasingly complex. These applications are subject to temporal and data constraints. They require a reliable and predictable implementation, since a mismatch during the system lifetime can be catastrophic. Therefore, a model is required to represent (in a deterministic way) the communications between the several parts of a real-time application. This section represents several studies that have been investigated to study real-time systems using data flow formalisms.

Cyclo-Static Data Flow

In [5, 6, 86], the authors consider acyclic CSDFG applications with a single input. They provide an analytical framework for extracting the timing parameters for the CSDFG actors. They compute the periodic tasks parameters using an estimation of the worst-case execution time. They also assume that each read and write has constant execution time. Each actor of CSDFG were scheduled as an implicit periodic task where its deadline is equal to its period. In our approach, we modelled cyclic and acyclic multi-periodic systems with single/multiple input. Moreover, periodic tasks admit deadlines which are not necessarily equal to their periods.

Affine Data Flow graph

Bouakaz et al. [20] propose an extension of the CSDFG model called Affine Data Flow graph (ADF in short). This latter is a time-triggered data flow model that explicitly represents each task execution during a complete iteration of the graph. Each task is associated with an activation clock and executed at each clock tick. Any couple clock of ticks in the network can be related by an affine function. Precedence constraints between the tasks' executions are respected since the affine functions ensure the correct order of clock ticks execution. In this context, the authors proposed an analysis framework to schedule the actors of the ADF graph as periodic tasks. However, modeling the execution behaviour using the clock ticks and the affine functions delays the process of finding a feasible schedule. In fact, representing the execution behaviour with clock ticks implicitly transforms the ADF into an equivalent HSDFG. In contrast, our approach modeled the communication within a multi-periodic system with a graph whose size is equal to the number of the system tasks.

Homogeneous Synchronous Data Flow Graph

In [60, 59], Hausmans et al. have presented a temporal analysis for fixed-priority scheduling. They considered that data flow applications are modelled as cyclic HSDFG/SDFGs. They compute an upper bound on the worst-case response time of each actor. They assume that the data flow graphs have a strictly periodic source while considering that the other actors have a data-driven operational semantics. Task timing properties were extracted based on its interference with the set of higher priority tasks executed on the same platform. This means that the computation of the timing parameters depends on the set of applications executed on the same platform. Their approaches differ in how to compute that interference. They used a period and

jitter characterization [60](difference between best-case and worst-case offsets), an enabling rate characterization [59] (describing for each time interval the minimum and maximum number of possible task enabling). Unlike their approach, we modeled the communications within a multi-rate system independently of the scheduling strategy or the other applications executed on the same platform.

In [2], Ali et al. proposed an algorithm for extracting the real-time properties of data flow applications. They extract timing parameters (such as offsets, periods and deadlines) of data flow applications with timing constraints (such as throughput and latency). Their approach converts each actor into a periodic arbitrary-deadline tasks. However, their method can only be applied on data flow applications modeled as HSDFGs which are less expressive than SDFGs. In addition, transforming a SDFG into an equivalent HSDFG requires the use of an unfolding process that duplicates each actor potentially an exponential number of times.

Synchronous Data Flow Graph

Recently, Singh et al. [109] extended the SDFG model in order to incorporate real-time properties. They associate to each actor a worst-case execution time. Moreover, they specify to each SDFG a single input and output actors with repetition factors equal to 1. Each SDFG was also characterized by a period and a relative deadline parameters. In order to schedule the SDFG upon preemptive uniprocessor, they developed an algorithm converting any SDFG task into a collection of equivalent 3-parameter sporadic tasks $\tau_i = (C_i, D_i, T_i)$ [94, 10]. These latter have the worst-case computational requirement exactly equal to the worst-case computational requirement of the SDFG task. In this thesis, we limit our study to periodic tasks. However, we consider that the real-time tasks can admit release dates. In addition, we modeled the communications, within a multi-periodic system using a particular class of SDFGs, without increasing the number of the system tasks.

4.1.1 Latency evaluation using data flow model

Other authors have modeled real-time applications using data flow graphs in order to compute their latency.

Processing Graphs Methods

Goddard [48] modeled a real-time data flow application using Processing Graphs Methods (PGM in short). In PGM, a system is expressed as directed graph where nodes correspond to processing functions and edges correspond to FIFO queues. Real-time requirements were verified using the Rate Based Execution (RBE in short) model. A RBE task t is specified by four parameters (x, y, d, e). The couple (x, y) represents the rate execution where x is the number of executions during the interval y. A task is also specified by its deadline d and its execution time e. The latency was measured as function of the source period. It was bounded according to the sum of the tasks execution times and the sink deadline. In our approach, the tasks periods are directly extracted from the real-time application. Hence, latency is computed according to the parameters of each couple of periodic communicating tasks which is more expressive in terms of the system's communications.

Homogeneous Synchronous Data Flow Graph

Moreira et al. [95] studied real-time scheduling of data flow programs using Homogeneous SDFG. In this latter, the consumption and production rates are equal to 1. Some tasks were added to the equivalent HSDFG in order to enforce the strictly periodic behaviour within a self-timed implementation. They defined latency as the time interval between two events. Latency between periodic tasks were computed only if their repetition factors are equal. However, our communication model allows to evaluate the latency between periodic tasks having different repetition factors. Despite SDFG is known to be convertible into HSDFG, Marcheti and Munier [90] have proven that this conversion is not polynomial.

In [36, 35], authors proposed an analytical framework for computing the periodic tasks' parameters for an application modeled as acyclic CSDFG, so that the tasks are scheduled in a strictly periodic manner. Accordingly, they compute the latency between the executions of two dependent tasks . Moreover, they defined the system latency as an iteration during which each task is executed according to its repetition factor.

4.1.2 Main difference

Existing techniques [48, 95, 5, 20, 60, 2, 109], transform each actor in a given data flow into (independent/dependent) real-time task(s). Tasks' parameters are specified so that the data flow temporal behavior is correctly reflected. Accordingly, scheduling data

flow graph is equivalent to schedule the set of real-time tasks. This can be done using existing analysis frameworks developed for real-time systems. In contrast, our approach consists in modeling communications within multi-periodic systems. We assume that the periodic tasks parameters and their communication scheme are user-provided. We describe how inter-task dependencies (at the job level) can be modeled using a particular class of SDFG [73]. Consequently, scheduling a multi-periodic system is equivalent to schedule a SDFG using techniques (methods) that have be developed within the SDFG community.

4.2 Scheduling strict periodic non-preemptive tasks

A real-time task is considered schedulable if its worst-case response time under a given scheduling algorithm is less than or equal to its deadline. Consequently, a set of tasks is schedulable under a given scheduling algorithm if all of its tasks are schedulable. Real-time scheduling theory yields several schedulability conditions in order to verify whether a set of tasks is schedulable under a given scheduling algorithm. Given a system and scheduling algorithm, a schedulability condition is termed sufficient, if all of the system tasks that are deemed schedulable according to the condition are in fact schedulable [33]. Similarly, a schedulability test is termed necessary, if all of the system tasks that are deemed unschedulable according to the condition are in fact unschedulable [33]. Moreover, a schedulability condition that is both sufficient and necessary is an exact schedulability condition.

In this section, we present some studies from the literature concerned with scheduling of applications modeled as real-time periodic tasks. More precisely, we emphasis on the scheduling problem for non-preemptive strictly periodic tasks. In this case, the successive executions of each task admit periodic starting dates that are separated by the task period.

4.2.1 Preemptive uniprocessor schedulability analysis

In this subsection, we present two well-known scheduling algorithms and their schedulability test upon preemptive uniprocessor. These algorithms are Rate Monotonic (RM in short) and Early Deadline First (EDF in short).

Rate Monotonic

The RM algorithm [85] is a preemptive scheduling with static priorities that are assigned to a set of independent periodic tasks according to their rates (periods). Higher execution priority is given to the task with the smaller period. The RM algorithm is considered as a fixed-priority algorithm, since the tasks periods are constant. The sufficient schedulability condition of an implicit-deadline periodic task set on uniprocessor using RM algorithm can be verified through the processor utilization, as follows:

Theorem 4.2.1 *A set of independent periodic tasks* $\mathcal{T} = \{t_1, \ldots, t_n\}$ *with implicit-deadlines is schedulable using RM if*

$$\sum_{t_i \in \mathcal{T}} u_i \leq n(2^{\frac{1}{n}} - 1).$$

Given an implicit-deadline periodic task set, RM is optimal among all fixed-priority algorithms. In other words, if a fixed-priority algorithm can schedule a set of implicit-deadline tasks, RM does it as well. However, RM is not optimal on uniprocessor for non-preemptive periodic tasks.

Early Deadline First

EDF algorithm [85] is a scheduling algorithm that schedules the executions of the tasks according to their absolute deadlines. Higher execution priority is given to task execution with earlier deadline. EDF algorithm is optimal for independent implicit-deadline periodic task set on preemptive uniprocessor. The schedulability condition can be verified through the processor utilization, as follows:

Theorem 4.2.2 *A set of independent tasks* $\mathcal{T} = \{t_1, \ldots, t_n\}$ *with implicit-deadlines is schedulable using EDF if and only if*

$$\sum_{t_i \in \mathcal{T}} u_i \leq 1.$$

Baruah et al. [11] provide a sufficient and necessary condition for scheduling a constrained-deadline periodic tasks using EDF on uniprocessor. However, this schedulability test is very time consuming, since it is know to be co-NP-complete in strong sense.

In this thesis, we focus our study on scheduling applications modeled as non-preemptive strictly periodic tasks.

4.2.2 Scheduling non-preemptive strictly periodic systems

In real-time systems, scheduling of preemptive periodic tasks have been extensively studied. Many uniprocessor schedulability conditions have been developed for several preemptive algorithms such as RM and EDF [85]. Goossens et al. [49] studied the problem of scheduling offset free systems. They propose a schedulability condition on multi-core platform using EDF algorithm. Their approach has polynomial time complexity. Unfortunately, George et al. [45] proved that the schedulability conditions in preemptive scheduling become, at best, necessary conditions in the non-preemptive case.

Non-preemptive scheduling problems must be studied since their resolution may offer advantages in term of schedulability. Several studies considered preemptive scheduling conditions in order to obtain the conditions for the non-preemptive versions. Taking into consideration the extra interference time caused by non-preemption, Baruah [9] followed the similar method in [49] and provided a sufficient schedulability condition for non-preemptive periodic tasks. Based on the problem windows analysis [8], Guan et al. [55] studied the problem of global non-preemptive fixed priority scheduling. In our approach, we consider a set of non-preemptive tasks.

Furthermore, other studies focused on non-preemptive scheduling. Jeffay et al. [65] showed that non-preemptive scheduling of real-time tasks is an NP-hard problem. They also derived a necessary and sufficient condition for the non-preemptive Earliest Deadline First scheduling algorithm (npEDF in short) when tasks are periodic with arbitrary offsets. Recently, Nasri et al. [97] showed that the latter condition is very pessimistic in the special case of harmonic tasks. They also studied the non-preemptive Rate Monotonic (npRM in short) for the same type of tasks and derived a sufficient condition of schedulability for both npRM and npEDF. The same authors [98] proposed a scheduling algorithm called “Precausious-RM” or “P-RM”. It is an online algorithm for scheduling periodic non-preemptive harmonic task set, with linear complexity. A relaxed algorithm derived from P-RM is also proposed thereafter, called “Lazy P-RM”. This algorithm improves the performance of P-RM when task execution times are short. In these works [65, 97, 98], non-preemptive periodic tasks were studied. However, in this thesis, we consider strictly periodic constraints. In the sequel, we present some studies from the literature concerned with scheduling of applications modeled as non-preemptive strictly periodic tasks.

Scheduling independent non-preemptive strictly periodic tasks

A more particular scheduling problem is the one with non-preemptive strictly periodic tasks. The non-preemptive scheduling problem is known to be NP-hard computational complexity [65]. Adding the strict periodicity constraint increases the problem complexity. Korst et al. [80] were the first to study the problem of scheduling a set of non-preemptive strictly periodic tasks. Their work was motivated by real-time video signal processing. The authors considered this problem on a minimum number of processors [79]. They showed that the problem is NP-complete in the strong sense, even in the case of a single processor, but that it is solvable in polynomial time if the periods and execution times are divisible. Thus, they proposed an approximation algorithm based on assigning tasks to processors according to some priority rule. Besides, they proposed a necessary and sufficient condition for the schedulability of two strictly periodic tasks. Later, Kermia et al [70] proposed a sufficient schedulability condition that generalizes the previous condition for a set of tasks. They imposed that the sum of the tasks execution times is less or equal to the greatest common divisors (GCDs) of task periods. In [92], Marouf and Sorel proved that this sufficient condition is very restrictive (pessimistic). They also gave a schedulability condition for implicit-deadline strictly periodic tasks and proposed an heuristic based on this condition. In contrast, our approach is not restricted to implicit-deadline tasks.

Eisenbrand et al. [37] considered the problem of scheduling non-preemptive strictly periodic tasks in order to minimize the number of processors. They studied tasks with harmonic periods, i.e., every period divides all other periods of greater value. In this case, they showed that there exists a 2-approximation for the minimization problem and that this result is tight. The same authors added additional constraints to the same problem with harmonic tasks and proposed an Integer Linear Programming (ILP) formulation and primal heuristics [38]. Marouf and Sorel [93] proposed a similar work and gave a scheduling heuristic. This latter is based on the constraint that the period of a candidate task has a multiple relationship with the task periods already scheduled. However, this thesis addresses a more challenging problem which is dealing with harmonic and non-harmonic periods.

Al-Sheikh et al. [1] used the Mixed Integer Linear Program (MILP) formulation in order to establish an exact framework for non-preemptive strictly periodic scheduling problem in Integrated Modular Avionics (IMA) platform. Afterwards, they proposed a best-response algorithm based on game theory [44]. This algorithm consists in

computing the largest possible change for all task execution times. Accordingly, they determined whether all tasks are schedulable upon a limited number of processors. Pira and Artigues [102] improved the best-response algorithm using a propagation mechanism and local optimizations. However, in our study we consider that the user provides all the tasks parameters.

Chen et al. [26] represented a strictly periodic task by its eigentask, i.e., a task that has the same period as the original task and an execution time equal to one. Assuming that a set of tasks is already scheduled on a given processor, they proposed a sufficient condition to determine whether a new task is schedulable on the same processor. Based on this condition, they developed a task assignment algorithm to allocate the tasks and compute an upper bound of the number of required processors . Afterwards, Chen et al. [25] adopted the idea of eigentask. They used a more efficient method to calculate the valid scheduling slots of an eigentask. In addition, they developed an heuristic that assigned the tasks to required processors. In their approach, the tasks priorities were not considered during the assignment step.

As mentioned previously, several studies [92, 37, 38, 93, 1, 102, 26, 25] investigated the scheduling problem of independent non-preemptive strictly periodic tasks. These approaches considered uniprocessor/multiprocessor scheduling problem. Tasks were assigned to the same processor either with harmonic or non-harmonic periods. However, none of these approaches considered the scheduling of dependent tasks set. In other words, these approaches do not take into account the communication constraints between the tasks executions. In this thesis, we tackle the mono-processor scheduling problem of strictly periodic tasks subject to communication constraints.

Scheduling dependent non-preemptive strictly periodic tasks

Other authors considered the scheduling problem of dependent non-preemptive strictly periodic tasks, taking into consideration data exchange (at the job level). Dependency between the tasks executions were modeled as precedence constraints.

Cucu and Sorel [30] considered the problem of multiprocessor scheduling of strictly periodic systems with communication requirements. They used a graph-based model to specify the precedence between the tasks executions through acyclic graphs [29]. In their approach, a precedence constraint between a pair of tasks (t_i, t_j) imposes that the period of t_i must be less or equal to that of its successor t_j. Communication (data

transfer) between tasks is only restricted to tasks that are executed at the same rate. In order to solve the scheduling problem, they proposed an heuristic and compared its performance to an exact "Branch & Bound" algorithm [82].

Based on the previous work, Kermia et al. [70] proposed a greedy heuristic for allocating and scheduling non-preemptive dependent strictly periodic systems onto multi-core platform. Their approach is composed of several algorithms. Assuming a scheduling frame whose length is equal to the system hyper-period, each task was duplicated ($\frac{hp}{T}$) times, where T is a task period and hp is the hyper-period. Arcs were added between duplicates in order to represent precedence constraints between the tasks executions. Duplicates were classified and assigned to processors according to their periods (in an increasing order). Their assignment favours mapping duplicates whose periods are equal or multiples on the same processor. Finally, an extension of the SynDEx heuristic [54] was applied onto the unrolled graph instead of the original one. Afterwards, the same authors [71] improved their assignment algorithm using a mixed sort which takes into account both the tasks periods (in an increasing order) and a priority level. In addition, data transfer can only be occurred between two tasks having the same or multiple periods. The heuristic performance was compared to an optimal "Branch and Cut" algorithm [88].

Contrary to the previous works [30, 70, 71], in this thesis, we limit our study to uniprocessor scheduling problem of strictly priodic communicating tasks. In our approach, tasks admit release dates and relative deadlines (user-provided). Communication scheme between the tasks execution is build according to these parameters (release dates and relative deadlines). Consequently, our framework allows communication between tasks with arbitrary periods. In order to solve this scheduling, we propose an exact (optimal) algorithm and three heuristics. These methods are able to schedule cyclic and acyclic systems on a single processor. Furthermore, our heuristics do not increase the number of tasks in order to schedule them on the same processor. In other words, we do not convert our original graph into an equivalent graph that models the communications between the tasks executions. This transformation is not polynomial and can be very time consuming [90]. Finally, tasks with non-harmonic periods can be assigned to the same processor.

4.3 Conclusion

In recent years, several models have been developed to study data flow graphs from a real-time point of view. This problem has been studied by researchers from both (real time and data flow) communities. Conversely to the approaches proposed in the literature, our approach consists in studying communications within a multi-periodic system from a data flow point of view.

On the other hand, scheduling applications modeled as non-preemtive strictly periodic tasks is very challenging, since this problem is known to be NP-complete in strong sense. Several approaches introduced in the literature considered the scheduling problem of an independent strictly periodic set of tasks. Unfortunately, only few authors tackled this scheduling problem with communication (precedence) constraints between the tasks executions. In order to meet this challenge, we propose in chapter 6 an optimal algorithm and three heuristics that solve the scheduling problem of strictly periodic systems subject to communication constraints.

The following chapter presents the first contribution of this thesis. This contribution consists in modeling the communications within a multi-periodic systems and evaluating their (worst-case) latencies using data flow formalisms.

Chapter 5

Real Time Synchronous Data Flow Graph (RTSDFG)

Nowadays, real-time applications are invading our daily life. These systems are highly critical, as a mismatch during the system lifetime can be catastrophic. In this context, one of the major challenges faced by academic and industrial environments is the efficient use of powerful and complex material, to provide optimal performance (producing the right outputs for given inputs) and meet the time constraints (producing outputs at the right time). These systems are usually multi-periodic, since their components communicate with each other at different rates (this is mainly due to their physical characteristic). For this reason, a deep analysis of communications between tasks of a multi-periodic system is required. Moreover, the estimation of parameters in a static manner, such as evaluating the latency between the system input and its corresponding outcome, is an important practical issue.

In this chapter, we seek to model the communications within multi-periodic systems using SDGF formalisms. Section 5.1 is dedicated to define the "Real-Time Synchronous Data Flow Graph". This latter describes the communication model within a multi-periodic system. Computing the latency between two periodic communicating tasks is developed in section 5.2. Several methods for evaluating the worst case system latency are explicitly described in Section 5.3. Experimental results of these methods are represented in Chapter 7. Conclusion is stated in Section 5.4.

5.1 modeling real time system communications using RTSDFG

This section shows that the set of communication constraints of a multi-periodic system can be modeled with a SDFG. In the beginning, we introduce the tasks model given by Liu and Layland [85]. Then, we define the precedence constraints between two periodic communicating tasks. Afterwards, we characterize the execution of these two tasks using a set of linear inequalities. Finally, we demonstrate that the communication constraints of a multi-periodic system can be modeled with a SDFG. The size of this latter is equivalent to the related communication graph.

5.1.1 Periodic tasks

We consider a set of tasks based on the model of Liu and Layland [85]. Each task t_i is characterized by an activation period T_i, an execution time C_i (this value is often evaluated from a worst case analysis of the system), a relative deadline D_i, and eventually a release date (date of the first activation) r_i. This implies that the successive executions of t_i, admit periodic release dates and deadlines, where the period is equal to T_i (see Figure 5.1).

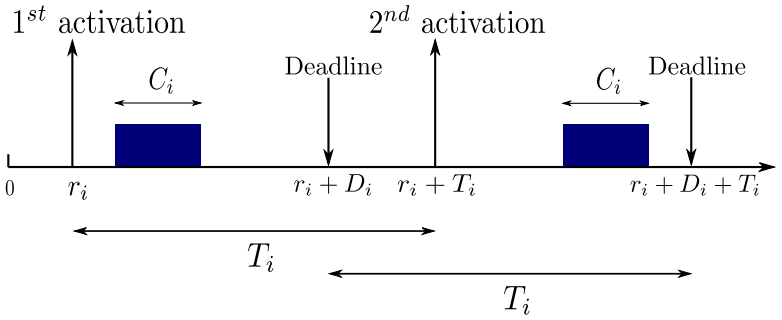


Fig. 5.1 Liu and Layland model

Therefore, the kth execution of t_i is executable if and only if its execution start date s_i^k is greater or equal to its release date

$$s_i^k \geq r_i^k$$
$$s_i^k \geq r_i + (k-1) \cdot T_i,$$

and its execution end date cannot exceed its deadline

$$s_i^k + C_i \leq r_i + (k-1) \cdot T_i + D_i.$$

5.1.2 Communication constraints

Most of real-time applications requires communication between tasks. This type of communication happens between an emitting task and a receiving one. The emitting task produces data that are consumed by the receiving task through a communication point. However, a receiving task cannot consume a piece of data unless this data has been sent by the emitting task. Thus, the execution of a receiving task should be preceded by the execution of an emitting task, which imposes precedence constraints between some executions.

Communication Scheme

In order to separate the tasks execution model from the communication scheme, we simplify the execution model of Liu and Layland. Therefore, we seek to increase for each task its execution time (as big as possible). We assume that each task starts its execution at its release date and it will be executed until it reaches its deadline. Thus, the execution start date of a given task and its release date coincide. Hence, the starting date of any task execution (t_i^k) is equal to

$$s_i^k = r_i + (k-1) \cdot T_i.$$

In other words, we build our communication scheme by assuming that data are available at the emitting task deadline. Moreover, these data can only be consumed at the beginning of the receiving task period. Accordingly, we assume that the execution time of each task (C_i) is equal to its relative deadline (D_i). **This assumption is only considered to define the communication scheme between the tasks executions. In the general case, this assumption does not prevent C_i to be smaller.**

As depicted in Figure 5.2, the task $t_i = (r_i, C_i, D_i, T_i)$ starts its first execution at its release date (r_i). This task is executed during C_i time units. Moreover, the second execution of t_i starts at $r_i + T_i$ such that T_i is the task period.

In order to describe the communication scheme, we consider two periodic communicating tasks $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$, such that t_i is the emitting

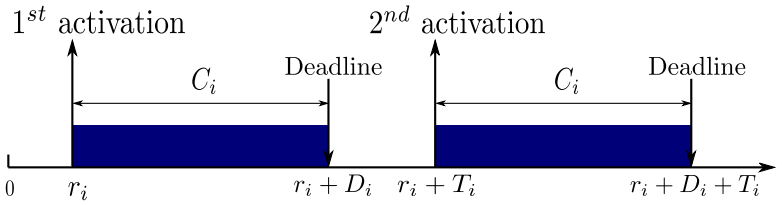


Fig. 5.2 Simplification of Liu and layland's model

task and t_j is the receiving one. Let ν_i and ν_j be a pair of strictly positive integers. There exists a precedence constraint between the ν_ith execution of t_i and the ν_jth execution of t_j if and only if the three following conditions are met:

1. The ν_jth execution of t_j cannot begin before the end of the ν_ith execution of t_i, thus

$$\begin{aligned} s_i^{\nu_i} + C_i &\leq s_j^{\nu_j} \\ r_i + (\nu_i - 1) \cdot T_i + D_i &\leq r_j + (\nu_j - 1) \cdot T_j \end{aligned} \tag{5.1}$$

2. There is no precedence constraint between the $(\nu_i + 1)$th execution of t_i and the ν_jth execution of t_j, since the ν_jth execution of t_j starts before the end of the $(\nu_i + 1)$th execution of t_i, hence

$$\begin{aligned} s_i^{\nu_i+1} + C_i &> s_j^{\nu_j} \\ r_i + \nu_i \cdot T_i + D_i &> r_j + (\nu_j - 1) \cdot T_j. \end{aligned} \tag{5.2}$$

3. There is no precedence constraint between the ν_ith execution of t_i and the $(\nu_j - 1)$th execution of t_j, since the $(\nu_j - 1)$th execution of t_j begins before the end of ν_ith execution of t_i, so

$$\begin{aligned} s_i^{\nu_i} + C_i &> s_j^{\nu_j - 1} \\ r_i + (\nu_i - 1) \cdot T_i + D_i &> r_j + (\nu_j - 2) \cdot T_j. \end{aligned} \tag{5.3}$$

Example 5.1.1 Let us consider $t_i = (r_i, C_i, D_i, T_i)$, $t_j = (r_j, C_j, D_j, T_j)$ two periodic communicating tasks. We suppose that $T_i = 30$, $T_j = 40$ time units, $C_i = D_i = 20$ time units, $C_j = D_j = 20$ time units, and their release dates are null ($r_i = r_j = 0$). Figure 5.3 illustrates the precedence constraints between the tasks executions. We notice the

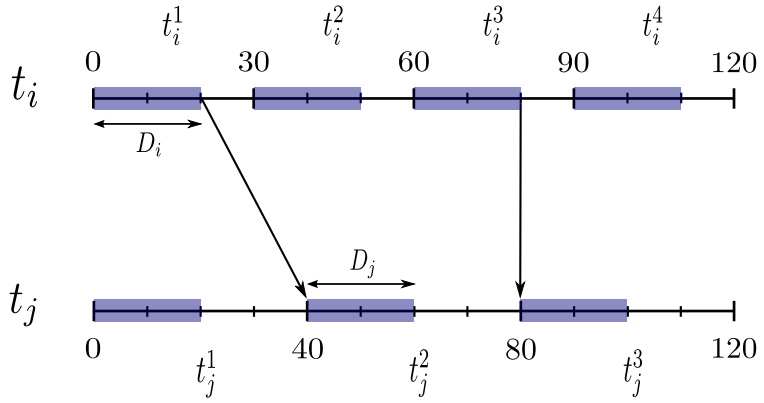


Fig. 5.3 Example of multi-periodic communication model

existence of a precedence constraint between the first execution of t_i and the second execution of t_j. In fact, equation (5.1) is fulfilled since the second execution of t_j did not start before the end of the first execution of t_i. Furthermore, we notice the absence of precedence constraints between t_i^1 and t_j^1 on one hand, and between t_i^2 and t_j^2 on the other hand. In both cases, the corresponding execution of t_j begins before the end of t_i execution. This means that equation (5.2) and (5.3) are also fulfilled.

The following lemma draws together the three inequalities mentioned before.

Lemma 5.1.1 *Let $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$ be two periodic communicating tasks, with t_i the emitting task and t_j the receiving one. Let ν_i and ν_j be a pair of strictly positive integers. There exists a precedence constraint between the ν_ith execution of t_i and the ν_jth execution of t_j if and only if:*

$$T_i \geq T_j - r_j + r_i + D_i + \nu_i \cdot T_i - \nu_j \cdot T_j > max\{0, T_i - T_j\} \tag{5.4}$$

Proof: Let us consider the inequalities (5.1), (5.2) and (5.3) which define the precedence constraints between the different executions of two communicating tasks. By substituting $T_j - r_j + r_i + D_i$ with $\mathcal{M}$, we obtain:

1. Condition 1 is equivalent to:

$$\mathcal{M} + \nu_i \cdot T_i - \nu_j \cdot T_j \leq T_i,$$

2. Condition 2 is equivalent to:

$$\mathcal{M} + \nu_i \cdot T_i - \nu_j \cdot T_j > 0,$$

3. Condition 3 is equivalent to:

$$\mathcal{M} + \nu_i \cdot T_i - \nu_j \cdot T_j > T_i - T_j.$$

By combining the three inequalities, we obtain

$$T_i \geq \mathcal{M} + \nu_i \cdot T_i - \nu_j \cdot T_j > max\{0, T_i - T_j\}.$$

□

Let us Consider the communication scheme illustrated in Figure 5.3. Equation (5.4) can be written according to the executions indexes (ν_i and ν_j):

$$30 \geq 40 + 20 + 30 \cdot \nu_i - 40 \cdot \nu_j > 0$$

We verify this inequality for the couples $(\nu_i, \nu_j) = (1,2)$ and $(3,3)$, which corresponds to the established precedence constraints between t_i^1 and t_j^2 and between t_i^3 and t_j^3.

5.1.3 From real time system to RTSDFG model

Based on the communication rules presented in the previous subsection, we prove in this subsection that the set of communication constraints between two periodic communicating tasks can be modeled using a SDFG buffer. The buffer production and consumption rates correspond to the tasks periods and its initial marking is computed with a close formula according to the tasks parameters. Accordingly, we deduce that communication constraints of a multi-periodic system can be modeled with a particular SDFG. The size of the latter is equal to the system communication graph size.

We note gcd_a the common period of two communicating tasks t_i and t_j. It is equal to the greatest common divider of tasks periods, $gcd_a = gcd(T_i, T_j)$. We assume for what follows, that all the tasks parameters are integers.

Theorem 5.1.1 *Let $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$ be two periodic tasks such that t_i communicates with t_j. The set of communication constraints between the executions of t_i and t_j can be modeled by a buffer $a = (t_i, t_j)$ with $z_i = T_i$, $z_j = T_j$ and $M_0(a) = T_j + \lambda - gcd_a$ with $\lambda = \left\lceil \frac{r_i - r_j + D_i}{gcd_a} \right\rceil \cdot gcd_a$.*

Proof: Let us consider a couple of periodic communicating tasks $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$ such that t_i is the emitting task and t_j the receiving one. Let (ν_i, ν_j) be a pair of strictly positive integers. We assume that there is a precedence

constraints between the ν_ith execution of t_i and the ν_jth execution of t_j. According to the lemma 5.1.1, ν_i and ν_j must verify the equation (5.4):

$$T_i \geq T_j - r_j + r_i + D_i + \nu_i \cdot T_i - \nu_j \cdot T_j > max\{0, T_i - T_j\}.$$

We can notice that all the terms of this inequality are divisible by the common period (gcd_a) except the following one: $r_i - r_j + D_i$. Hence, by substituting $r_i - r_j + D_i$ by $\lambda = \left\lceil \frac{r_i - r_j + D_i}{gcd_a} \right\rceil \cdot gcd_a$, we obtain

$$T_i \geq T_j + \lambda + \nu_i \cdot T_i - \nu_j \cdot T_j > max\{0, T_i - T_j\}.$$

Consequently, all the terms of this inequality are divisible by gcd_a. By subtracting gcd_a from the second term of this inequality, we obtain

$$T_i > T_j + \lambda - gcd_a + \nu_i \cdot T_i - \nu_j \cdot T_j \geq max\{0, T_i - T_j\}. \tag{5.5}$$

On the other hand, let $a = (t_i, t_j)$ be a buffer and $M_0(a)$ its initial marking. Let (ν_i, ν_j) be a couple of strictly positive integers. According to Theorem 3.2.1, there exists a precedence constraint between the ν_ith execution of t_i and the ν_jth execution of t_j if and only if:

$$z_i \; > \; M_0(a) \; + \; z_i \cdot \nu_i \; - \; z_j \cdot \nu_j \geq max\{0, z_i - z_j\}. \tag{5.6}$$

When we compare inequalities (5.5) and (5.6), we deduce that the set of communication constraints between the executions of t_i and t_j can be modeled by a buffer $a = (t_i, t_j)$. The buffer production and consumption rates are respectively equal to T_i and T_j and its initial marking $M_0(a)$ is equal to $T_j + \lambda - gcd_a$. □

Let us return to the example illustrated in Figure 5.3, the common period is equal to $gcd_a = gcd(30, 40) = 10$ time units. According to Theorem 5.1.1, all the communication constraints between the tasks executions can be modeled with a buffer $a = (t_i, t_j)$. The buffer production and consumption rates are respectively equal to $z_i = T_i = 30$, $z_j = T_j = 40$ and its initial marking is equal to $M_0(a) = T_j + \lambda - gcd_a = 40 + \lceil \frac{0+20}{10} \rceil \cdot 10 - 10 = 50$ (see Figure 5.4).

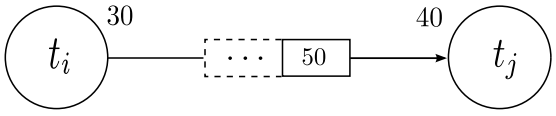


Fig. 5.4 The Buffer $a = (t_i, t_j)$ that models the communication constraints between the tasks executions of the example illustrated in Figure 5.3.

Corollary 5.1.1 *Let $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$ be two strictly periodic communicating tasks. Let $a = (t_i, t_j)$ be a SDFG buffer modeling the communications between the tasks executions. Let (ν_i, ν_j) be a pair of strictly positive integers. Let $s_i^{\nu_i}$ and $s_j^{\nu_j}$ be respectively the execution starting dates of $t_i^{\nu_i}$ and $t_j^{\nu_j}$. Communication constraints represented by the buffer $a = (t_i, t_j)$ are fulfilled, if*

$$\forall \nu_i \geq 1,\ s_i^{\nu_i} \in [r_i + (\nu_i - 1) \cdot T_i,\ r_i + D_i - C_i + (\nu_i - 1) \cdot T_i] \text{ and,}$$
$$\forall \nu_j \geq 1,\ s_j^{\nu_j} \in [r_j + (\nu_j - 1) \cdot T_j,\ r_j + D_j - C_j + (\nu_j - 1) \cdot T_j].$$

Proof: We consider two periodic communicating tasks $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$ which communicate according to the communication scheme defined previously. According to Theorem 5.1.1, there exists a buffer $a = (t_i, t_j)$ which models the set of precedence constraints induced by the communications between the executions of t_i and t_j. Accordingly, the data are only available at the deadline of the emitting task and the consumption of these data can only be done at the begin of the receiving task period. Communications between the tasks executions depend on the release dates and the deadlines of t_i on one hand, and they only depend on the release dates of t_j on the other hand.

We assume that a precedence constraint (induced by the buffer a) exists between the ν_ith execution of t_i and the ν_jth execution of t_j. This means that data are available at the deadline of $t_i^{\nu_i}$ which is equal to $r_i + D_i + (\nu_i - 1) \cdot T_i$. These data are consumed at the release date of $t_j^{\nu_j}$ which is equal to $r_j + (\nu_j - 1) \cdot T_j$.

Furthermore, let $s_i^{\nu_i}$ and $s_j^{\nu_j}$ be respectively the starting dates of $t_i^{\nu_i}$ and $t_j^{\nu_j}$ respectively, such that

$$s_i^{\nu_i} \in [r_i + (\nu_i - 1) \cdot T_i,\ r_i + D_i - C_i + (\nu_i - 1) \cdot T_i]$$
$$s_j^{\nu_j} \in [r_j + (\nu_j - 1) \cdot T_j,\ r_j + D_j - C_j + (\nu_j - 1) \cdot T_j]$$

Consequently, the end of the execution of the emitting task ($t_i^{\nu_i}$) cannot exceed its deadline, since $s_i^{\nu_i} \leq r_i + D_i - C_i + (\nu_i - 1) \cdot T_i$. Moreover, the receiving task cannot

begin its execution before its release date, since $s_j^{\nu_j} \geq r_j + (\nu_j - 1) \cdot T_j$. Hence, the precedence constraint between $t_i^{\nu_i}$ and $t_j^{\nu_j}$ is fulfilled. In this case, we can deduce that the set of communication constraints represented by the buffer $a = (t_i, t_j)$ is fulfilled. □

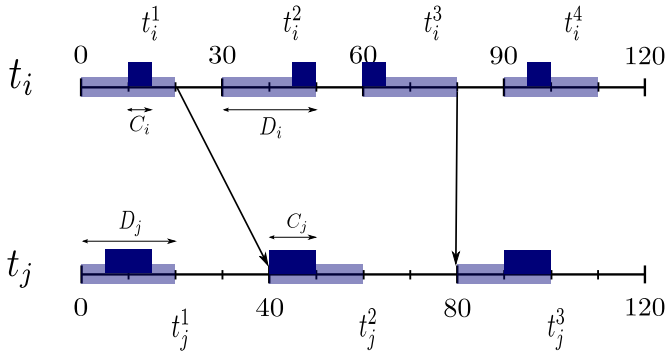


Fig. 5.5 An example of scheduling which respects the communication constraints between the tasks $t_i = (0, 5, 20, 30)$ and $t_j = (0, 10, 20, 40)$.

Example 5.1.2 Consider two periodic communicating tasks $t_i = (0, 5, 20, 30)$ and $t_j = (0, 10, 20, 40)$. Let $S_i = (10, 45, 60, 95)$ and $S_j = (5, 40, 90)$ be respectively the starting dates of t_i and t_j' executions. Since the tasks are executed between their release dates and their deadlines, we can deduce that the communications constraints between the tasks' executions are respected (see Figure 5.5).

Corollary 5.1.2 *Let $\mathcal{N}$ be a set of periodic communicating tasks. Let $\mathcal{H}$ be the corresponding communication graph. The precedence constraints induced by the communications are modeled with a particular class of SDFG.*

We named the particular class of SDFG that models all the precedence constraints induced by the communications of a multi-periodic communicating tasks set as "Real-Time Synchronous Data Flow Graph" (RTSDFG in short). In this particular class of SDFG, the buffer production and consumption rates are no longer representing the amount of data produced and consumed at each task execution. These rates correspond to the tasks periods while the initial marking corresponds to the delay between the tasks.

Therefore, we deduce that our problem is equivalent to a problem that takes as input a set of periodic tasks with a corresponding communication graph. Modeling this problem returns a particular class of SDFG which called RTSDFG. The size of the resulting graph is equal to the size of the communication graph. The RTSDFG models all the communication constraints between the tasks' executions. Table 5.1

presents a set of periodic tasks $\mathcal{N} = \{t_1, t_2, t_3\}$. The tasks communication relationships $\mathcal{E} = \{(t_1, t_2), (t_1, t_3), (t_2, t_1), (t_3, t_2)\}$ are depicted in Figure 5.6. The RTSDFG associated to this multi-periodic system is depicted in Figure 5.7.

t_i	r_i	C_i	D_i	T_i
t_1	0	5	20	30
t_2	20	10	20	40
t_3	5	5	10	20

Table 5.1 Periodic tasks parameters

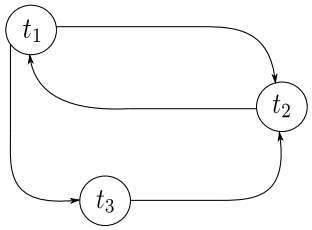


Fig. 5.6 Communication graph $\mathcal{H}$

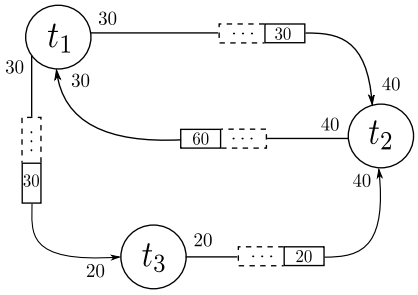


Fig. 5.7 The RTSDFG that models the communications between the tasks' executions of the multi-periodic system presented in Table 5.1 and Figure 5.6

In the previous parts of this chapter, we proposed a general and intuitive communication model between multi-periodic tasks. Now, the question that needs to be raised is:

How to compute the latency of a multi-periodic system using a Real-Time Synchronous Data Flow Graph?

The next part of this chapter answers this question in two steps. First, we compute the latency between two periodic communicating tasks, then we compute the worst case system latency using several approaches.

5.2 Evaluating latency between two periodic communicating tasks

In this section, we define the latency between two periodic communicating tasks. We show that the latency between the tasks executions is calculated with a closed formula

according to the tasks parameters. Moreover, we express the maximum and minimum latency between two communicating tasks with closed formulas. Finally, we prove that the minimum and maximum latency can be bounded according to the tasks periods.

In order to evaluate the worst-case system latency, we assume that each task starts its execution at its release date and it will be executed until its deadline. According to this assumption, evaluating the latency between two communicating executions, is equivalent to compute the duration between the data availability and its consumption.

5.2.1 Definitions

Definition 5.2.1 *Latency is the time gap between the moment that a stimulation appears and the moment that its reaction begins or ends. Evaluating the latency $\mathcal{L}$ between two executions of periodic communicating tasks, is equivalent to compute the duration between the execution end date of the emitting task and the execution start date of the receiving task (since we assume that each task starts its execution at its release date and ends at its deadline).*

Definition 5.2.2 *Let t_i and t_j be a couple of periodic communicating tasks. We consider the set $E(t_i,t_j)$ of couples $(\nu_i,\nu_j) \in \mathbb{N}^{*^2}$, such that there exists a precedence constraint between $t_i^{\nu_i}$ and $t_j^{\nu_j}$.*

Definition 5.2.3 *Let $t_i^{\nu_i}$ and $t_j^{\nu_j}$ be a couple of executions with $(\nu_i,\nu_j) \in E(t_i,t_j)$. Hence, $t_j^{\nu_j}$ cannot start its execution before the end of $t_i^{\nu_i}$. We denote by $\mathcal{L}(t_i^{\nu_i},t_j^{\nu_j})$ the latency between the ν_ith execution of t_i and the ν_jth execution of t_j. This latency can be computed as follows:*

$$\begin{aligned}\mathcal{L}(t_i^{\nu_i},t_j^{\nu_j}) &= s_j^{\nu_j} - (s_i^{\nu_i} + C_i) \\ &= r_j + (\nu_j - 1)\cdot T_j - (r_i + (\nu_i - 1)\cdot T_i + D_i).\end{aligned}$$

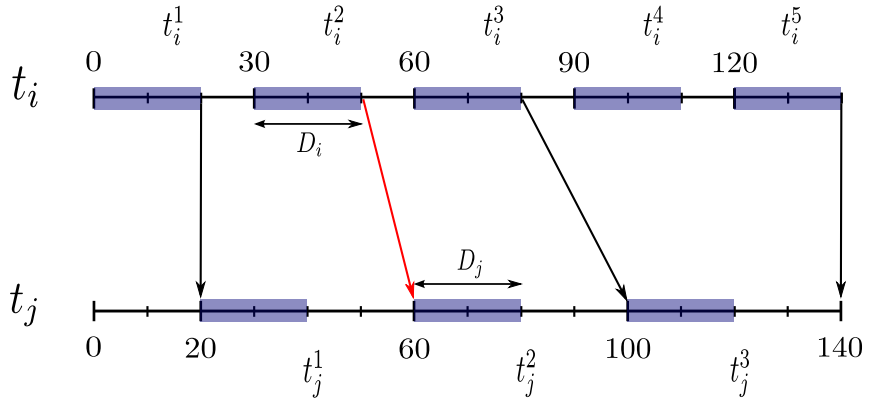


Fig. 5.8 Example of communication scheme between two periodic communicating tasks.

Example 5.2.1 Consider two periodic communicating tasks $t_i = (0,5,20,30)$ and $t_j = (20,10,20,40)$ such that t_i is the emitting task and t_j the receiving one. Figure 5.8 illustrates the communication scheme between the tasks executions. We can notice the existence of a precedence (communication) constraint between the t_i^2 and t_j^2. Latency between these execution can be computed as follows:

$$\begin{aligned}\mathcal{L}(t_i^2, t_j^2) &= 20 + (2-1)\cdot 40 - (0 + (2-1)\cdot 30 + 20) \\ &= 10 \text{ time units.}\end{aligned}$$

Computing latency between two periodic communicating tasks

Lemma 5.2.1 *Let t_i and t_j be two periodic communicating tasks, with t_i the emitting task and t_j the receiving one. Let $(\nu_i, \nu_j) \in E(t_i, t_j)$ be a couple of strictly positive integers. The latency between $t_i^{\nu_i}$ and $t_j^{\nu_j}$ is equal to:*

$$\mathcal{L}\left(t_i^{\nu_i}, t_j^{\nu_j}\right) = r_j - r_i - k \cdot gcd_a - T_j + T_i - D_i, \tag{5.7}$$

where $gcd_a = \gcd(T_i, T_j)$ and $k \in \{k_{min}, \ldots, k_{max}\}$.

Proof: We assume that there is a precedence constraint between $t_i^{\nu_i}$ and $t_j^{\nu_j}$. Furthermore, the ν_ith execution of t_i ends at $s_i^{\nu_i} + C_i$ and the ν_jth execution of t_j starts at $s_j^{\nu_j}$.

By assumption, $s_i^{\nu_i} = r_i + (\nu_i - 1) \cdot T_i$ and $s_j^{\nu_j} = r_j + (\nu_j - 1) \cdot T_j$. Therefore, the latency between the ν_ith execution of t_i and the ν_jth execution of t_j is equal to

$$\begin{aligned}\mathcal{L}\left(t_i^{\nu_i}, t_j^{\nu_j}\right) &= s_j^{\nu_j} - (s_i^{\nu_i} + C_i) \\ &= r_j + (\nu_j - 1) \cdot T_j - r_i - (\nu_i - 1) \cdot T_i - D_i.\end{aligned}$$

Then, we consider a RTSDFG buffer $a = (t_i, t_j)$. The buffer production and consumption rates are respectively equal to T_i and T_j, and its initial marking is equal to $M_0(a) = T_j + \lambda - gcd_a$. According to Theorem 5.1.1, the buffer a models all the communication constraints between tasks executions. Therefore, Lemma 3.2.1 of Chapter 3 establishes a relation between the indexes of two adjacent tasks connected by a buffer. According to this lemma, there exists an integer $k \in \{k_{min}, \ldots, k_{max}\}$ such that $\nu_j = \frac{T_i \cdot \nu_i - k \cdot gcd_a}{T_j}$, with $k_{min} = \frac{\max\{0, T_i - T_j\} - M_0(a)}{gcd_a}$ and $k_{max} = \frac{T_i - M_0(a)}{gcd_a} - 1$.

Hence, the latency between ν_ith execution of t_i and the ν_jth execution of t_j can be computed as follows:

$$\begin{aligned}\mathcal{L}\left(t_i^{\nu_i}, t_j^{\nu_j}\right) &= r_j + (\nu_j - 1) \cdot T_j - r_i - (\nu_i - 1) \cdot T_i - D_i \\ &= r_j + \left(\frac{T_i \cdot \nu_i - k \cdot gcd_a}{T_j} - 1\right) \cdot T_j - r_i - (\nu_i - 1) \cdot T_i - D_i \\ &= r_j - r_i - k \cdot gcd_a - T_j + T_i - D_i\end{aligned}$$

□

Let us consider the communication scheme depicted in Figure 5.8. Let $a = (t_i, t_j)$ be the RTSDFG buffer that models the communication constraints between the tasks executions. The buffer production and consumption rates are respectively equal to 30 and 40 and its initial marking is equal to 30. Therefore,

$$\begin{aligned}k_{min} &= \frac{\max\{0, 30 - 40\} - 30}{10} = -3, \\ k_{max} &= \frac{30 - 30}{10} - 1 = -1.\end{aligned}$$

Hence, we can deduce that the latency between $t_i^{\nu_i}$ and $t_j^{\nu_j}$ such that $(\nu_i, \nu_j) \in E(t_i, t_j)$, can be computed as follows:

$$\begin{aligned}\mathcal{L}\left(t_i^{\nu_i}, t_j^{\nu_j}\right) &= r_j - r_i - k \cdot gcd_a - T_j + T_i - D_i \\ &= 20 - 0 - k \cdot 10 - 40 + 30 - 20 \\ &= -10 \cdot (k+1),\end{aligned}$$

with $k \in \{-3, -2, -1\}$.
We apply this result to compute the latency between t_i^2 and t_j^2 with $k = -2$,

$$\mathcal{L}(t_i^2, t_j^2) = -10 \cdot (-2+1) = 10 \text{ time units.}$$

5.2.2 Maximum latency between two periodic communicating tasks

Definition 5.2.4 *We define $\mathcal{L}_{max}(t_i, t_j)$ as the maximum latency between the executions of two periodic communicating tasks,*

$$\mathcal{L}_{max}(t_i, t_j) = \max_{(\nu_i, \nu_j) \in E(t_i, t_j)} \mathcal{L}\left(t_i^{\nu_i}, t_j^{\nu_j}\right).$$

Theorem 5.2.1 *Let t_i and t_j be two periodic communicating tasks, with t_i the emitting task and t_j the receiving one. The maximum latency between executions of these tasks is equal to:*

$$\mathcal{L}_{max}(t_i, t_j) = r_j - r_i - \max\{0, T_i - T_j\} + \lambda - gcd_a + T_i - D_i. \tag{5.8}$$

Proof: We consider two periodic tasks t_i and t_j. Let $(\nu_i, \nu_j) \in E(t_i, t_j)$ be a couple of strictly positive integers. According to Lemma 5.2.1, the latency between $t_i^{\nu_i}$ and $t_j^{\nu_j}$ can be written as follows:

$$\mathcal{L}_{(\nu_i, \nu_j)}(t_i, t_j) = r_j - r_i - k \cdot gcd_a - T_j + T_i - D_i.$$

In order to evaluate the maximum latency between t_i and t_j, we substitute k with its minimal value $k_{min} = \frac{\max\{0, T_i - T_j\} - M_0(a)}{gcd_a}$ [90]. In this case, we obtain $\mathcal{L}_{max}(t_i, t_j)$ equals to:

$$r_j - r_i - \left(\frac{\max\{0, T_i - T_j\} - M_0(a)}{gcd_a} \right) \cdot gcd_a - T_j + T_i - D_i.$$

Furthermore, $M_0(a) = T_j + \lambda - gcd_a$. Hence, $\mathcal{L}_{max}(t_i, t_j)$ equals

$$r_j - r_i - \max\{0, T_i - T_j\} + (T_j + \lambda - gcd_a) - T_j + T_i - D_i.$$

Finally, we deduce that the maximum latency is equal to

$$r_j - r_i - \max\{0, T_i - T_j\} + \lambda - gcd_a + T_i - D_i.$$

□

In our example, depicted in Figure 5.8, the maximum latency between t_i and t_j is equal to

$$\begin{aligned}
\mathcal{L}_{max}(t_i, t_j) &= r_j - r_i - \max\{0, T_i - T_j\} + \lambda - gcd_a + T_i - D_i \\
&= 20 - 0 - \max\{0, 30 - 40\} + \lceil \frac{0 - 20 + 20}{10} \rceil \cdot 10 - 10 + 30 - 20 \\
&= 20 \text{ time units.}
\end{aligned}$$

This value corresponds exactly to the duration between the end of t_i^3 and the beginning of t_j^3.

Corollary 5.2.1 *Let t_i and t_j be two periodic communicating tasks, with t_i the emitting task and t_j the receiving one. the maximum latency value is bounded as follows:*

$$T_i - \max\{0, T_i - T_j\} - gcd_a \leq \mathcal{L}_{max}(t_i, t_j) < T_i - \max\{0, T_i - T_j\} \tag{5.9}$$

Proof: Let t_i and t_j two periodic communicating tasks such that t_i communicates with t_j. According to Theorem 5.2.1, the maximum latency between the executions of t_i and t_j can be computed as follows:

$$\mathcal{L}_{max}(t_i, t_j) = r_j - r_i - \max\{0, T_i - T_j\} + \lambda - gcd_a + T_i - D_i,$$

with $\lambda = \lceil \frac{r_i - r_j + D_i}{gcd_a} \rceil \cdot gcd_a$.

On the other hand, λ can be bounded as follows:

$$r_i - r_j + D_i \leq \lambda < r_i - r_j + D_i + gcd_a$$

Furthermore,

$$0 \leq \lambda - (r_i - r_j + D_i) < gcd_a,$$
$$T_i - \max\{0, T_i - T_j\} - gcd_a \leq \lambda - (r_i - r_j + D_i) + T_i - \max\{0, T_i - T_j\} - gcd_a < T_i - \max\{0, T_i - T_j\}.$$

Hence, we can deduce that

$$T_i - \max\{0, T_i - T_j\} - gcd_a \leq \mathcal{L}_{max}(t_i, t_j) < T_i - \max\{0, T_i - T_j\}$$

□

Let us reconsider our example in Figure 5.8. Computing the maximum latency lower and upper bounds between the tasks t_i and t_j can be done in the following way:

$$30 - \max\{0, 30 - 40\} - 10 \leq \mathcal{L}_{max}(t_i, t_j) < 30 - \max\{0, 30 - 40\}$$
$$20 \leq \mathcal{L}_{max}(t_i, t_j) < 30.$$

5.2.3 Minimum latency between two periodic communicating tasks

Definition 5.2.5 *We denote by $\mathcal{L}_{min}(t_i, t_j)$ the minimum latency between the executions of two communicating tasks,*

$$\mathcal{L}_{min}(t_i, t_j) = \min_{(\nu_i, \nu_j) \in E(t_i, t_j)} \mathcal{L}(t_i^{\nu_i}, t_j^{\nu_j}).$$

Theorem 5.2.2 *Let t_i and t_j be two periodic communicating tasks , with t_i the emitting task and t_j the receiving one. The minimum latency between executions of these two tasks is equal to:*

$$\mathcal{L}_{min}(t_i, t_j) = r_j - r_i + \lambda - D_i.$$

Proof: We consider two periodic tasks t_i and t_j. Let $(\nu_i, \nu_j) \in E(t_i, t_j)$ a couple of strictly positive integers. According to the Lemma 5.2.1, the latency between $t_i^{\nu_i}$ and $t_j^{\nu_j}$ can be written as follows:

$$\mathcal{L}\left(t_i^{\nu_i}, t_j^{\nu_j}\right) = r_j - r_i - k \cdot gcd_a - T_j + T_i - D_i.$$

In order to be able to evaluate the minimum latency between t_i and t_j, we substitute k with its maximal value $k_{max} = \frac{T_i - M_0(a)}{gcd_a} - 1$ [90]. In this case, we obtain

$$\begin{aligned}\mathcal{L}_{min}(t_i,t_j) &= r_j - r_i - k_{max} \cdot gcd_a - T_j + T_i - D_i \\ &= r_j - r_i - (\frac{T_i - M_0(a)}{gcd_a} - 1) \cdot gcd_a - T_j + T_i - D_i \\ &= r_j - r_i + M_0(a) + gcd_a - T_j - D_i.\end{aligned}$$

Furthermore, we have $M_0(a) = T_j + \lambda - gcd_a$. Hence, we deduce that $\mathcal{L}_{min}(t_i,t_j)$ is equal to

$$r_j - r_i + \lambda - D_i.$$

□

Let us consider our communication example depicted in Figure 5.8, the minimum latency between the tasks executions can be compute as follows:

$$\begin{aligned}\mathcal{L}_{min}(t_i,t_j) &= r_j - r_i + \lambda - D_i \\ &= 20 - 0 + \lceil \frac{0 - 20 + 20}{10} \rceil \cdot 10 - 20 \\ &= 0 \text{ time units.}\end{aligned}$$

This value corresponds exactly to the duration between the end the first execution of t_i and the begging of the first execution of t_j.

Corollary 5.2.2 *Let t_i and t_j be two periodic communicating tasks, with t_i the emitting task and t_j the receiving one. the minimum latency value can be bounded as follows:*

$$0 \leq \mathcal{L}_{min}(t_i,t_j) < gcd_a \tag{5.10}$$

Proof: Let t_i and t_j two periodic communicating tasks such that t_i communicates with t_j. According to the Theorem 5.2.2, the minimum latency between tasks executions can be computed as follows:

$$\mathcal{L}_{min}(t_i,t_j) = r_j - r_i + \lambda - D_i,$$

with $\lambda = \lceil \frac{r_i - r_j + D_i}{gcd_a} \rceil \cdot gcd_a$.

On the other hand, λ can be bounded as follows:

$$r_i - r_j + D_i \leq \lambda < r_i - r_j + D_i + gcd_a.$$

Hence, we can deduce that

$$0 \leq \mathcal{L}_{min}(t_i, t_j) < gcd_a.$$

□

We apply this result on our example depicted in figure 5.8,

$$0 \leq \mathcal{L}_{min}(t_i, t_j) < 10.$$

5.3 Evaluating the worst-case system latency

This section presents methods for latency computation of a given system. First, we define the worst-case system latency. Then, we present an exact method, that uses all the precedence relationships between tasks' executions, to compute the latency value. Finally, we propose two algorithms that calculate the system latency upper and lower bounds. These algorithms transform the RTSDFG into an equivalent graph having the same size of the RTSDFG. In this graph, the weight of each arc connecting two adjacent nodes corresponds to the time required to transfer the data between the periodic tasks. In order to evaluate the worst-case latency of a multi-periodic system, we limit our study in this thesis to live, consistent and connected acyclic RTSDFGs. Moreover, we compute the system latency using a linear algorithm for computing the longest path length in a directed acyclic graph.

5.3.1 Definition

Definition 5.3.1 *Latency is the delay between the stimulation and response. In other words, entire system latency is a time gap between the system input and the connected outcome. In our study, we define the worst-case system latency as the longest time gap between the system input task executions and the output ones.*

As we mentioned before, in order to evaluate the worst-case system latency, we assume that each task of the system starts its execution at its release date and ends at its deadline.

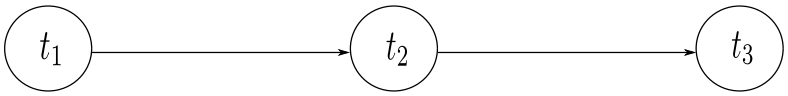


Fig. 5.9 Path $pth = \{t_1, t_2, t_3\}$ which corresponds to the communication graph of three periodic tasks.

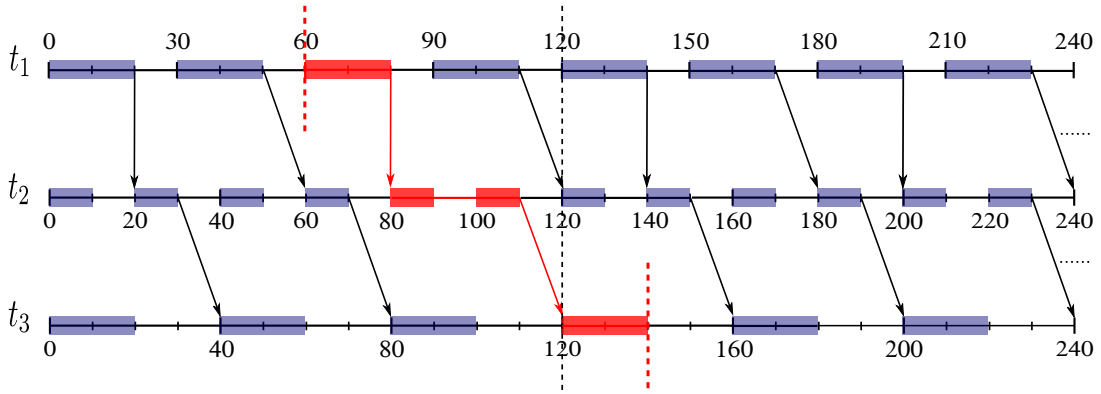


Fig. 5.10 Communication and execution model of the path $pth = \{t_1, t_2, t_3\}$ depicted in Figure 5.9.

Example 5.3.1 Let $t_1 = (0, 10, 20, 30), t_2 = (0, 5, 10, 20)$ and $t_3 = (0, 10, 20, 40)$ be three periodic communicating tasks. Figure 5.9 represents the tasks communication graph. Figure 5.10 illustrates all the communications constraints between the tasks executions of the path $pth = \{t_1, t_2, t_3\}$. We consider t_1 as the system input and t_3 the system output. According to our communication model, the data produced by the first execution of t_1 is consumed by the second execution of t_2, which in turn produces data that is consumed by the second execution of t_3. We summarize all communications constraints between the input and the output tasks' executions as follows: $pth_1 = \langle t_1^1 \rightarrow t_2^2 \rightarrow t_3^2 \rangle$, $pth_2 = \langle t_1^2 \rightarrow t_2^4 \rightarrow t_3^3 \rangle$ and $pth_3 = \langle t_1^3 \rightarrow t_2^5 \rightarrow t_2^6 \rightarrow t_3^4 \rangle$. The time required to cover these executions paths is respectively equal to 60, 70 and 80 time units. This communications and executions pattern is repeated infinitely. Hence, we can deduce that the system worst-case latency is equal to $\max\{60, 70, 80\} = 80$ time units. This value corresponds to the time gap between the beginning of the third execution of t_1 and the end of the the fourth execution of t_3.

Computing the latency of applications that are executing infinitely is a problematic. Actually, the size of the graph, which represents the communication constraints of these applications, is bounded. The question that needs to be raised at this stage is:

how to represent the communications constraints between the tasks executions in a finite way?
Next part answers this question by introducing an exact pricing algorithm that represents these constraints in a finite way and evaluates the system worst-case latency.

5.3.2 Exact pricing algorithm

We present in this part an exact pricing algorithm that computes the worst-case latency between a system input and its corresponding output. This algorithm computes the system latency using all communication constraints between the tasks executions. It is therefore necessary to explicitly represent these communications. Our approach is inspired from the *expansion* [96] which is previously detailed in Section 3.2 of Chapter 3. This technique consists on transforming a SDFG into an equivalent HSDFG that explicitly models all the precedence constraints between the tasks executions.

Dependency between executions of two communicating tasks

Let t_i and t_j be two periodic communicating tasks. Let $a = (t_i, t_j)$ be the RTSDFG buffer that models all the communication constraints between the tasks' executions. Let $(\nu_i, \nu_j) \in E(t_i, t_j)$ be a pair of strictly positive integers, such that there is a precedence constraint between $t_i^{\nu_i}$ and $t_j^{\nu_j}$. The buffer marking after the ν_jth execution of t_j and the ν_ith execution of t_i is equal to

$$M_0(a) + \nu_i \cdot T_i - \nu_j \cdot T_j.$$

Our purpose is to describe the dependency between $t_i^{\nu_i}$ and $t_j^{\nu_j+d}$ with $d \in \mathbb{N}^*$. We seek to find out how many times t_j can be executed using the remaining amount of tokens in the buffer $(M_0(a) + \nu_i \cdot T_i - \nu_j \cdot T_j)$. In fact, t_j cannot be executed unless there is a least T_j tokens in the buffer. This means if the remaining amount of tokens is less than T_j, $t_j^{\nu_j+d}$ depends on the $t_i^{\nu_i+1}$.

We denote by *dep* the number of executions succeeding (following) the ν_jth execution of t_j and which are totally dependent from the ν_ith execution of t_i. This number of execution can be computed, according to the buffer consumption rate and the buffer marking after the $t_i^{\nu_i}$ and $t_j^{\nu_j}$ executions, as follows:

$$dep = \lfloor \frac{M_0(a) + \nu_i \cdot T_i - \nu_j \cdot T_j}{T_j} \rfloor$$

On the other hand, since there is a precedence constraint between $t_i^{\nu_i}$ and $t_j^{\nu_j}$, ν_i and ν_j verify the following equation

$$T_i > M_0(a) + \nu_i \cdot T_i - \nu_j \cdot T_j \geq \max\{0, T_i - T_j\}$$

Hence, *dep* can be bounded as follows:

$$\lfloor \frac{T_i}{T_j} \rfloor \geq \lfloor \frac{M_0(a) + \nu_i \cdot T_i - \nu_j \cdot T_j}{T_j} \rfloor \geq \lfloor \frac{\max\{0, T_i - T_j\}}{T_j} \rfloor \tag{5.11}$$

We distinguish two cases:

– If $T_i \leq T_j$:

Equation (5.11) can be written in the following form,

$$\lfloor \frac{T_i}{T_j} \rfloor \geq dep \geq 0.$$

Actually, since $T_i \leq T_j$, we can deduce that the number of executions that follow $t_j^{\nu_j}$ and which totally depend from $t_i^{\nu_i}$ is null ($dep = 0$).

– If $T_i > T_j$:

We can bound *dep* according to the tasks periods,

$$\lfloor \frac{T_i}{T_j} \rfloor \geq \lfloor \frac{M_0(a) + \nu_i \cdot T_i - \nu_j \cdot T_j}{T_j} \rfloor \geq \lfloor \frac{T_i - T_j}{T_j} \rfloor.$$

Hence, we deduce that $dep \in \{\lfloor \frac{T_i}{T_j} \rfloor - 1, \lfloor \frac{T_i}{T_j} \rfloor\}$. We can also deduce that at most ($\lceil \frac{T_i}{T_j} \rceil$) executions of t_j can totally depend on t_i execution.

RTSDFG transformation

Consider $\mathcal{N} = \{t_1, \ldots, t_n\}$ a set of periodic communicating tasks and $\mathcal{H}$ the corresponding communication graph. Let $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ be the RTSDFG that models all communications constraints between the tasks executions of this system. The main idea here is to evaluate the latency using a finite equivalent graph $\mathcal{G}'$. This latter describes exactly the precedence constraints of the original graph. Transforming a RTSDFG $\mathcal{G}$ into $\mathcal{G}' = (\mathcal{T}', \mathcal{A}')$ can be done as follows:

- Each $t_i \in \mathcal{T}$ is substituted by R_i (repetition factor of t_i) tasks denoted by $t_i^1, \ldots, t_i^{R_i}$ such that for any $k \in \{1, \ldots, R_i\}$ and $\alpha > 0$, the αth execution of t_i^k corresponds

to the $((\alpha-1)\cdot R_i+k)$th execution of t_i. Tasks $t_i^1,\ldots,t_i^{R_i}$ are called duplicates of t_i. We note that the duplicates have the same parameters than the original task.

- For each pair of strictly positive integers $(\nu_i,\nu_j)\in E(t_i,t_j)$, we add an arc between $t_i^{\nu_i}$ and $t_j^{\nu_j}$. Based on the *expansion* [96] which is previously detailed in Section 3.2 of Chapter 3, arcs are added between duplicates only if the following conditions are met:

$$\begin{cases}\nu_j=\lfloor\frac{M_0(a)+T_i\cdot(\nu_i-1)}{T_j}\rfloor+1 & \text{if } T_i>T_j \text{ and } \nu_i\geq\frac{-M_0(a)}{T_i}+1\\ \nu_i=\lceil\frac{T_j\cdot\nu_j-M_0(a)}{T_i}\rceil & \text{if } T_i\leq T_j \text{ and } \nu_j>\frac{M_0(a)}{T_j}\end{cases} \tag{5.12}$$

- If $T_i>T_j$, dependency between tasks' executions is described by adding arcs between $t_i^{\nu_i}$ and the set of executions $\{t_j^{\nu_j+d}, d\in\{1,\ldots,dep\}\}$ with $dep=\lfloor\frac{M_0(a)+\nu_i\cdot T_i-\nu_j\cdot T_j}{T_j}\rfloor$.

- The weight of each arc in $\mathcal{A}'$ is equal to

$$\omega(t_i^{\nu_i},t_j^{\nu_j})=r_j+(\nu_j-1)\cdot T_j-r_i-(\nu_i-1)\cdot T_i. \tag{5.13}$$

  This value corresponds to the duration between the beginning of the ν_ith execution of t_i and the beginning of the ν_jth execution of t_j.

Algorithm 1 describes the transformation of a RTSDFG $\mathcal{G}=(\mathcal{T},\mathcal{A})$ into an equivalent graph $\mathcal{G}'=(\mathcal{T}',\mathcal{A}')$. The Duplication of each task $t_i\in\mathcal{T}$, according to its repetition factor, is represented in line 5. Once the tasks are duplicated, arcs are added between duplicates in two cases:

1. For describing precedence between executions: we connect each couple of duplicates $(t_i^{\nu_i},t_j^{\nu_j})$ with an arc if there exists a precedence relationship between the tasks executions. This is given in line 8.

2. For describing dependency between executions: in order to evaluate the system latency, some arcs must be added to express dependency between executions. These arcs are added if there is a precedence relationship between two executions ($t_i^{\nu_i}$ and $t_j^{\nu_j}$) such that $T_i>T_j$. Line 12 verifies if there is any other executions of t_j which totally depend on the ν_ith execution of t_i. The number of these dependent executions *dep* is computed in line 11 according to the buffer consumption rate and the remaining amount of tokens in the buffer after the ν_ith execution of t_i and the ν_jth execution of t_j.

Algorithm 1 Transformation of a RTSDFG into a equivalent graph $\mathcal{G}^{'}$

1: **Input**: RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$
2: **Output**: $\mathcal{G}^{'} = (\mathcal{T}^{'}, \mathcal{A}^{'})$
3: pair of strictly positive integers: (ν_i, ν_j)
4: arc weight : $\omega(t_i^{\nu_i}, t_j^{\nu_j})$
5: **substitute** each task $t_i \in \mathcal{T}$ by the set of duplicates $\{t_i^1, \ldots, t_i^{R_i}\}$
6: **for all** $a = (t_i, t_j) \in \mathcal{A}$ **do**
7: **for all** $(\nu_i, \nu_j) \in E(t_i, t_j)$ **do**
8: **add** arc between $t_i^{\nu_i}$ and $t_j^{\nu_j}$
9: $\omega(t_i^{\nu_i}, t_j^{\nu_j}) \leftarrow r_j + (\nu_j - 1) \cdot T_j - r_i - (\nu_i - 1) \cdot T_i$
10: **if** $T_i > T_j$ **then**
11: $dep = \lfloor \frac{M_0(a) + T_i \cdot \nu_i - T_j \cdot \nu_j}{T_j} \rfloor$
12: **if** $dep > 0$ **then**
13: **add** arcs from $t_i^{\nu_i}$ to $\{t_j^{\nu_j + d}, d \in \{1, \ldots, dep\}\}$
14: $\omega(t_i^{\nu_i}, t_j^{\nu_j + d}) \leftarrow r_j + (\nu_j - 1 + d) \cdot T_j - r_i - (\nu_i - 1) \cdot T_i$
15: **end if**
16: **end if**
17: **end for**
18: **end for**

Finally, lines 9 and 14 in Algorithm 1 compute the weight of each arc in the resulting graph $\mathcal{G}^{'}$.

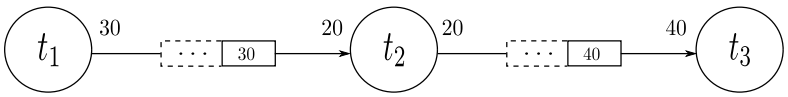


Fig. 5.11 RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ modeling the communication constraints between the tasks' executions of the example represented in Figure 5.10.

Example 5.3.2 Let us consider the RTSDFG depicted in Figure 5.11 which models the communication constraints between the tasks executions presented in Figure 5.10. The RTSDFG repetition vector is equal to $R = [4, 6, 3]$. Figure 5.12 illustrates the equivalent graph obtained by applying Algorithm 1 on this RTSDFG. This graph represents all the precedence and dependency constraints induced by the buffers of the original graph.

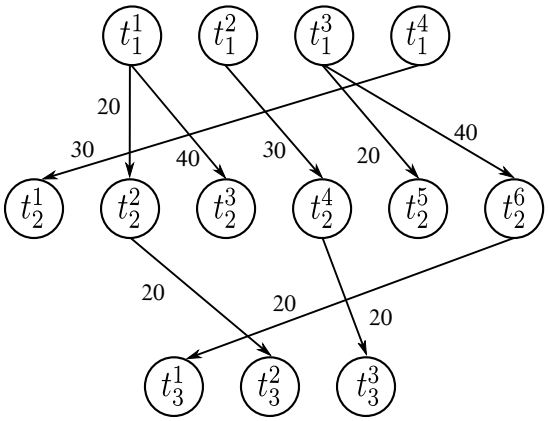


Fig. 5.12 Graph $\mathcal{G}^{'} = (\mathcal{T}^{'}, \mathcal{A}^{'})$ that models precedence and dependency constraints between the tasks executions of the RTSDFG depicted in Figure 5.11.

First, we substitute each task by its duplicates (according to its repetition factor). In this example, t_1, t_2 and t_3 are respectively substituted by $\{t_1^1, t_1^2, t_1^3, t_1^4\}$, $\{t_2^1, t_2^2, t_2^3, t_2^4, t_2^5, t_2^6\}$ and $\{t_3^1, t_3^2, t_3^3\}$. Then, precedence and dependency constraints are modeled by adding arcs between the tasks' duplicates. In our example, there exists a buffer between t_1 and t_2 such that $T_1 > T_2$. According to equation (5.12), we connect each couple of duplicates $(t_1^{\nu_1}, t_2^{\nu_2})$ if

$$\nu_2 = \lfloor \frac{30 + 30 \cdot (\nu_1 - 1)}{20} \rfloor + 1,$$

such as $\left(t_1^1, t_2^2\right)$. The arc weight which connects these duplicates is equal to

$$0 + 20 \cdot (2-1) - 0 - 30 \cdot (1-1) = 20 \text{ (eqution 5.13)}.$$

In addition, as $T_1 > T_2$, we compute the number of executions which may totally depend on the first execution of t_1. This can be done by computing the value of *dep* which is equal to

$$dep = \lfloor \frac{30 + 30 \cdot 1 - 20 \cdot 2}{20} \rfloor = 1 \text{ with } \nu_1 = 1 \text{ and } \nu_2 = 2.$$

We describe this dependency by adding an arc from t_1^1 to t_2^{2+1} whose weight is equal to

$$0 + 20 \cdot (3-1) - 0 - 30 \cdot (1-1) = 40.$$

On the other hand, t_2 is connected to t_3 with a buffer where $T_2 < T_3$. By applying equation (5.12), we connect each couple of duplicates $(t_3^{\nu_3}, t_2^{\nu_2})$ if

$$\nu_2 = \lceil \frac{40 \cdot \nu_3 - 40}{20} \rceil \text{ for } \nu_3 > \frac{40}{40},$$

such as the couple $\left(t_2^2, t_3^2\right)$. The arc weight which connects these duplicates is equal to

$$0 + 40 \cdot (2-1) - 0 - 20 \cdot (2-1) = 20.$$

Latency computation

After transforming the RTSDFG $\mathcal{G}$ into $\mathcal{G}^{'}$ which models all the precedence and dependency constraints between the tasks executions, we evaluate the worst-case system latency using $\mathcal{G}^{'}$. More precisely, worst-case latency evaluation is done by finding the longest path in the equivalent graph $\mathcal{G}^{'}$. The steps of this evaluation are detailed in Algorithm 2.

Definition 5.3.2 *We denote by* $\mathcal{P} = \{t \mid prec(t) = \emptyset\}$ *the set of nodes without predecessor. In addition, we denote by* $\mathcal{S} = \{t \mid succ(t) = \emptyset\}$ *the set of nodes without successor.*

Algorithm 2 Computation of the worst-case system latency

1: **Input**: equivalent graph $\mathcal{G}^{'} = (\mathcal{T}^{'}, \mathcal{A}^{'})$
2: **Output**: $\mathcal{L}_{\mathcal{G}}$ worst-case system latency
3: **add** a super source: *src*
4: **add** a super destination : *dst*
5: **for all** $t \in \mathcal{P}$ **do**
6: **add** arc between *src* and t
7: $\omega(src, t) \leftarrow 0$
8: **end for**
9: **for all** $t \in \mathcal{S}$ **do**
10: **add** arc between t and *dst*
11: $\omega(t, dst) \leftarrow$ relative deadline of t (D_t)
12: **end for**
13: $\mathcal{L}_{\mathcal{G}} \leftarrow$ longest path length between *src* and *dst*

Algorithm 2 describes the worst-case latency computation of a RTSDFG. This algorithm takes as input the equivalent graph resulting from the transformation

presented in Algorithm 1 and returns as output the value of the worst-case system latency. First, we add to $\mathcal{G}'$ two virtual nodes: a super source and a super destination (lines 3 and 4). Then, we connect *src* to each node $t \in \mathcal{P}$ with an arc whose weight is null (from line 5 to 8). We connect each node $t \in \mathcal{S}$ to *dst* with an arc whose weight is equal to the relative deadline of t (from line 9 to 12). Adding these virtual nodes does not influence the time behavior of the system. Finally, we find the worst-case latency value for the entire system by computing the length of the longest path between the virtual nodes *src* and *dst* (line 13).

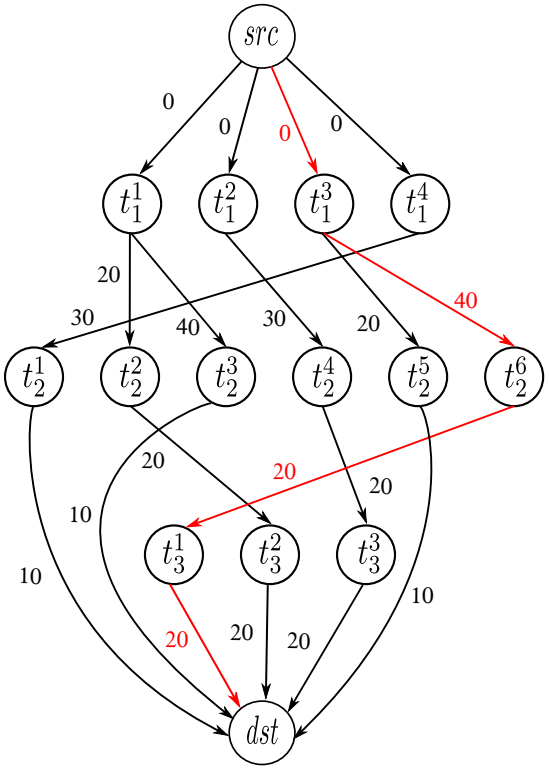


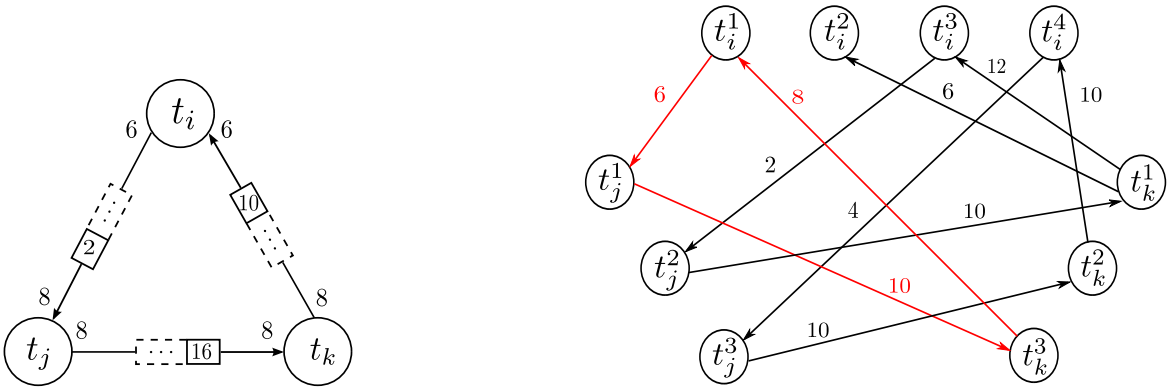
Fig. 5.13 Computation of the Worst-case latency of the RTSDFG depicted in Figure 5.11.

In order to evaluate the worst-case latency of the RTSDG in Figure 5.11, we use Algorithm 2 which takes as input the equivalent graph $\mathcal{G}'$ depicted in Figure 5.12. We add two virtual nodes (*src*/*dst*) to the equivalent graph. Then, we connect *src* to each duplicate without predecessor $\mathcal{P} = \{t_1^1, t_1^2, t_1^3, t_1^1\}$ with arcs whose weights are null. On the other hand, we add arcs between each duplicate without successor $\mathcal{S} = \{t_2^1, t_2^3, t_2^5, t_3^1, t_3^2, t_3^3\}$ and *dst*. Their weights are equal to the tasks relative deadline ($D_2 = 10$ and $D_3 = 20$). Figure 5.13 depicts the resulting graph. Finally, in order to

evaluate the latency of the entire system, we compute the length of the longest path between *src* and *dst* which is equal to 80 time units. This value matches the result obtained previously in Figure 5.10.

Cyclic RTSDFGs

We mentioned previously that evaluating the worst-case system latency using our approach is restricted to acyclic RTSDFG. In this paragraph, we give a clear illustration of how this evaluation is no longer possible for a cyclic RTSDFG. Let us consider a RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ that contains a cycle $\Omega = (t_i, t_j, t_k)$. We assume that the ν_ith execution of t_i will influence the ν_kth execution of t_k. As there is a buffer between t_k and t_i, the same execution will affect the $(\nu_i + \epsilon)$th execution of t_i such that $\epsilon \in \mathbb{N}^*$. Hence, an execution occurred in a cycle can loop infinitely because the beginning of a task execution in a cycle depends on its own end.



(a) Cyclic RTSDFG. (b) Equivalent graph $\mathcal{G}'$

Fig. 5.14 A cyclic RTSDFG with its equivalent graph that models the precedence and dependence constraints between the tasks executions.

Figure 5.14a depicts a cyclic RTSDFG composed of three tasks t_i, t_j and t_k. The equivalent graph $\mathcal{G}' = (\mathcal{T}', \mathcal{A}')$ which models the precedence and dependency constraints between the tasks' executions is represented in Figure 5.14b. The weight of each arc in $\mathcal{T}'$ is strictly positive and it corresponds to the time gap between the beginnings of two executions in relation. As we can see, there exists precedence constraints between $t_i^1 \rightarrow t_j^1$, $t_j^1 \rightarrow t_k^3$ and $t_k^3 \rightarrow t_i^1$. This implies that the equivalent graph contains a cycle whose arcs' weights are strictly positive. Therefore, we can deduce that the worst-case latency computation is no longer possible, since the computation of the longest path in the equivalent graph $\mathcal{G}'$ does not converge.

Computation of the longest path

The NP-hardness of the longest path problem for arbitrary graphs can be shown using the reduction from Hamiltonian path problem. However, computing the longest path in directed acyclic graphs (DAG in short) has a linear solution using a topological sorting. It can be computed by applying a linear time algorithm for shortest paths in a DAG [21] in which every weight is replaced by its negation. The complexity of this algorithm is $\mathcal{O}(n+m)$ where n is the number of nodes and m is the number of arcs of the graph. However, the complexity of transforming the RTSDFG into an equivalent graph is not polynomial. The computation of this complexity is detailed in Section 7.2 of Chapter 7.

In the next part of this chapter, we present two algorithms that compute respectively the upper and lower bounds of the system latency. The computation of these bounds does not require the use of the equivalent graph which explicitly represents the precedence and the dependency constraints between the tasks executions.

5.3.3 Upper bound

In order to evaluate the upper and lower bounds of the system latency, our approach consists on transforming the RTSDFG into a weighted graph. The size of this latter is equal to the size of the original graph. In addition, the weight of each arc in the weighted graph corresponds to the time required to transfer the data between two task in the worst (or best) case scenario.

Let us consider the RTSDFG buffer $a = (t_i, t_j)$ with $z_i = T_i$, $z_j = T_j$ and $M_0(a)$ its initial marking. We assume that a models all the communication constraints between the two periodic communicating tasks $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$. According to our communication model presented in section 5.1.2, data are only available at the deadline of the emitting task t_i. Moreover, these data can only be consumed at the release date of the receiving task t_j. On the other hand, the latency between two communicating tasks is equal to the time gap between the deadline of t_i and the release date of t_j. According to Theorem 5.2.1, the maximum latency between the executions of t_i and t_j can be written in the following format:

$$\mathcal{L}_{max}(t_i, t_j) = r_j - r_i + \lambda + T_i - max\{0, T_i - T_j\} - gcd_a - D_i,$$

where $\lambda = \left\lceil \frac{r_i - r_j + D_i}{gcd_a} \right\rceil \cdot gcd_a$.

In order to compute the time required to transfer the data between t_i and t_j in the worst case scenario, two cases should be studied:

1. If $T_i \leq T_j$,

$$\mathcal{L}_{max}(t_i,t_j) = r_j - r_i + \lambda + T_i - gcd_a - D_i.$$

After an execution of t_i, t_j can be executed at most one time. Hence, the time required to ensure the data transfer between these tasks, in the worst case scenario, is equal to

$$\begin{aligned} D_i + \mathcal{L}_{max}(t_i,t_j) &= D_i + r_j - r_i + \lambda + T_i - gcd_a - D_i \\ &= r_j - r_i + \lambda + T_i - gcd_a. \end{aligned}$$

In our example in Figure 5.10 on page 75, $t_2 = (0,5,10,20)$ communicates with $t_3 = (0,10,20,40)$ where $T_2 < T_3$ and $gcd_a = gcd(20,40) = 20$. Hence, the time needed for the data transfer from t_2 to t_3, in the worst case scenario, is equal to

$$\begin{aligned} D_2 + \mathcal{L}_{max}(t_2,t_3) &= r_3 - r_2 + \lambda + T_2 - gcd_a \\ &= 0 + \lceil \frac{0+10}{20} \rceil \cdot 20 + 20 - 20 \\ &= 20 \text{ time units.} \end{aligned}$$

2. If $T_i > T_j$,

$$\mathcal{L}_{max}(t_i,t_j) = r_j - r_i + \lambda + T_j - gcd_a - D_i.$$

After an execution of t_i, t_j can be executed at most $\lceil \frac{T_i}{T_j} \rceil$ times. This means that at most $\lceil \frac{T_i}{T_j} \rceil$ executions of t_j are totally depend on the execution of t_i. Accordingly, the time separating the beginnings of the first and the last executions of t_j that depend on t_i is equal to $(\lceil \frac{T_i}{T_j} \rceil - 1) \cdot T_j$ time units. Hence, in order to ensure the data transfer from t_i to t_j, the time required in the worst case scenario is equal to

$$D_i + \mathcal{L}_{max}(t_i,t_j) + (\lceil \frac{T_i}{T_j} \rceil - 1) \cdot T_j = r_j - r_i + \lambda - gcd_a + \lceil \frac{T_i}{T_j} \rceil \cdot T_j.$$

In the same example on page 75, $t_1 = (0,10,20,30)$ and $t_2 = (0,5,10,20)$ are communicating with $T_1 > T_2$ and $gcd_a = gcd(30,20) = 10$. Therefore, an execution

of t_1 will affect at most $\lceil \frac{T_1}{T_2} \rceil = \lceil \frac{30}{20} \rceil = 2$ executions of t_2. Accordingly, the time separating the beginnings of the first and the last executions of t_2 that totally depend on the execution of t_1 is equal to $(2-1) \cdot 20$ time units. In the worst case, the time needed to transfer the data between these tasks is equal to

$$\begin{aligned} D_i + \mathcal{L}_{max}(t_i, t_j) + (\lceil \frac{T_i}{T_j} \rceil - 1) \cdot T_j &= r_j - r_i + \lambda - gcd_a + \lceil \frac{T_i}{T_j} \rceil \cdot T_j \\ &= 0 + \lceil \frac{0+20}{10} \rceil \cdot 10 - 10 + 2 \cdot 20 \\ &= 50 \text{ time units.} \end{aligned}$$

In the next paragraph, we propose an algorithm covering the two cases below.

Latency upper bound computation

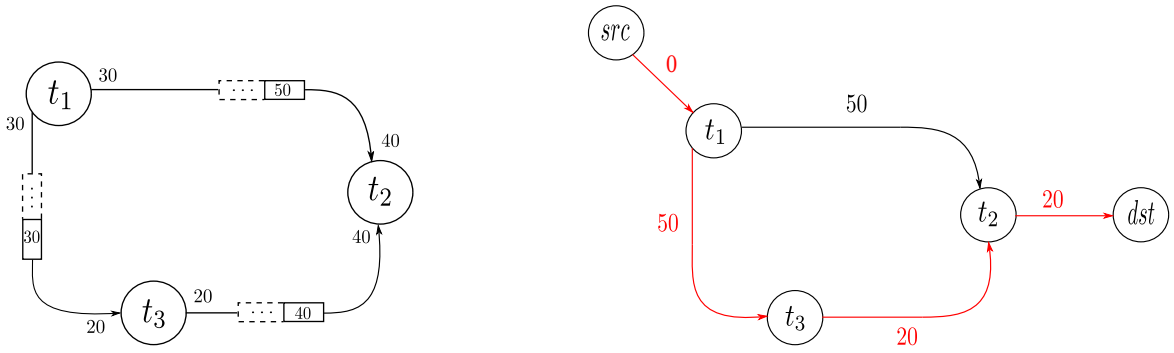
Algorithm 3 From RTSDFG to weighted graph $\mathcal{G}_{max}$

1: **Input**: RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$
2: **Output**: Weighted graph $\mathcal{G}_{max}$
3: arc weight: $\omega(e)$
4: **substitute** each task $t_i \in \mathcal{T}$ by a node t_i
5: **substitute** each buffer $a = (t_i, t_j) \in \mathcal{A}$ by an arc $e = (t_i, t_j)$
6: **for all** $a = (t_i, t_j) \in \mathcal{A}$ **do**
7: **if** $T_i \leq T_j$ **then**
8: $\omega(e) \leftarrow r_j - r_i + \lambda + T_i - gcd_a$
9: **else**
10: $\omega(e) \leftarrow r_j - r_i + \lambda + \lceil \frac{T_i}{T_j} \rceil \cdot T_j - gcd_a$
11: **end if**
12: **end for**

Algorithm 3 converts a RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ into a weighted graph $\mathcal{G}_{max}$ having the same size. Each task $t_i \in \mathcal{T}$ is represented by a node (line 4) and each buffer $a = (t_i, t_j)$ is represented by an arc $e = (t_i, t_j)$ (line 5). The arc weight $\omega(e)$ corresponds to the time required for data transfer from t_i to t_j in the worst case scenario. This weight is computed in two different cases according to the values of the buffer production and consumption rates. The two cases are respectively represented in line 8 and 10.

In order to evaluate the upper bound of the system latency, we add to the resulting graph $\mathcal{G}_{max}$ two virtual nodes *src* and *dst*. Then, we connect *src* to each node without

predecessor with an arc whose weight is null. We also connect each node without successor to *dst* with an arc whose weight is equal to the relative deadline of the source task. We evaluate the latency upper bound by computing the longest path between the virtual nodes (*src* and *dst*).



(a) Acyclic RTSDFG. (b) Weighted graph $\mathcal{G}_{max}$

Fig. 5.15 Example of latency upper bound computation using the weighted graph $\mathcal{G}_{max}$

Consider the RTSFG in Figure 5.15a that models the communication constraints between the executions of $t_1 = (0, 10, 20, 30)$, $t_2 = (0, 10, 20, 40)$ and $t_3 = (0, 5, 10, 20)$. Figure 5.15b presents the weighted graph $\mathcal{G}_{max}$. In this graph, the weight of each arc connecting two nodes corresponds to the time required to transfer the data between these nodes in the worst case scenario. For example, the time required to ensure the communication between $(t_1, t_3), (t_1, t_2)$ and (t_3, t_2) is respectively equal to $50, 50$ and 20 time units.

Let $\mathcal{P} = \{t_1\}$ be the set of nodes that do not have any predecessor. In addition, let $\mathcal{S} = \{t_2\}$ be the set of nodes that do not have any successor. Two super nodes (*src* / *dst*) are added to $\mathcal{G}_{max}$. We connect *src* to t_1 with an arc whose weight is null, and t_2 to *dst* with an arc whose weight is equal to $D_2 = 20$ time units. In order to evaluate the latency upper bound, we compute the length of the longest path between *src* and *dst* which is equal to 90 time units.

5.3.4 Lower bound

Let $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$ be two periodic communicating tasks. Let $a = (t_i, t_j)$ be RTSDFG buffer that models all the communication constraints between the tasks executions. According to Theorem 5.2.2, the minimum latency between the tasks executions is computed as follows:

$$\mathcal{L}_{min}(t_i, t_j) = r_j - r_i + \lambda - D_i.$$

Therefore, the time required to transfer the data from t_i to t_j, in the best-case scenario, is equal to

$$\begin{aligned} D_i + \mathcal{L}_{min}(t_i, t_j) &= D_i + r_j - r_i + \lambda - D_i \\ &= r_j - r_i + \lambda. \end{aligned}$$

In our example in Figure 5.10 on page 75, $t_2 = (0, 5, 10, 20)$ communicates with $t_3 = (0, 10, 20, 40)$ where $gcd_a = gcd(20, 40) = 20$. Hence, the time needed for the data transfer from t_2 to t_3, in the best case scenario, is equal to

$$\begin{aligned} D_2 + \mathcal{L}_{min}(t_2, t_3) &= r_3 - r_2 + \lceil \frac{r_2 - r_3 + D_2}{gcd_a} \rceil \cdot gcd_a \\ &= 0 + \lceil \frac{0 + 10}{20} \rceil \cdot 20 \\ &= 20 \text{ time units.} \end{aligned}$$

Latency lower bound computation

In order to evaluate the lower bound of the system latency, we transform a RTSDFG into a weighted graph $\mathcal{G}_{min}$. This transformation uses the same principle of Algorithm 1, which transform the RTSDFDG into a weighted graph $\mathcal{G}_{max}$. By contrast, in this case, the weight of each arc connecting two communicating nodes (t_i, t_j) is equal to

$$\omega(e) = r_j - r_i + \lambda.$$

We also add two virtual nodes (src/dst) to $\mathcal{G}_{min}$ (see Figure 5.16). We evaluate the latency lower bound by computing the longest path between the virtual nodes.

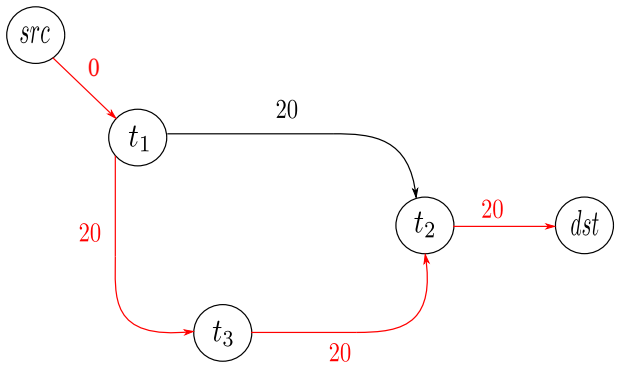


Fig. 5.16 Latency lower bound computation of the RTSFDG depicted in Figure 5.15a using the weighted graph $\mathcal{G}_{min}$.

Figure 5.16 depicts the weighted graph $\mathcal{G}_{min}$ of the RTSDFG depicted in Figure 5.15a. For instance, the time required for data transfer from $t_1 = (0, 10, 20, 30)$ to $t_2 = (0, 10, 20, 40)$, in the best case scenario, is equal to

$$\begin{aligned} D_1 + \mathcal{L}_{min}(t_1, t_2) &= r_2 - r_1 + \lambda \\ &= 0 + \lceil \frac{20}{20} \rceil \cdot 10 \\ &= 20 \text{ time units.} \end{aligned}$$

In order to evaluate the latency lower bound, we compute the longest path between src and dst. In this example, latency lower bound is equal to 60 time units.

5.4 Conclusion

In this chapter, we demonstrated that the communication within a multi-periodic system can be modeled by a RTSDFG. The size of this latter is equal to the communication graph size. Moreover, we expressed the latency between two communicating tasks using a closed formula. In the general case, we evaluated the latency of a system modeled by a RTSDFG using an exact pricing algorithm. In addition, we proposed two polynomial-time algorithms that compute respectively the latency upper and lower bounds. The three algorithms were applied on several instances of RTSDFGs, the experimental results are detailed in Section 7.2 of Chapter 7.

As the set of communication constraints within a multi-periodic system can be modeled by a RTSDFG, the following chapter addresses the scheduling problem of strictly periodic communicating tasks on a single processor.

Chapter 6

Scheduling strictly periodic systems with communication constraints

Preemptive real-time systems have received considerable research efforts in the real-time community, compared to non-preemptive systems. However, these latter are usually adopted in practical hard-real real time systems where missing deadlines may lead to catastrophic situations. Moreover, many industrial applications (e.g. avionic applications) require strictly periodic tasks. This means that the tasks successive executions admit periodic starting dates separated by the task period. This strict periodicity is usually required at the system inputs and outputs, such as the sensors and the actuators of a control/command applications.

In many hard real-time systems, communication is becoming increasingly indispensable. Due to the tasks strict deadlines that must be met, communications between the tasks executions are implemented in a completely deterministic manner. Our approach models the communication requirements as precedence constraints between tasks. More precisely, this approach consists of modeling the communications within multi-periodic systems using RTSDFG formalisms.

In this chapter, we consider the mono-processor scheduling problem for non-preemptive strictly periodic systems subject to communication constraints. Section 6.1 presents the non-preemptive strictly periodic model and its characteristics. In addition, it recalls the schedulability condition of two independent tasks on the same processor. Section 6.2 defines the SDFG schedules, especially the periodic ones that combine an explicit description of starting dates and an easy implementation. Section 6.3 introduces the schedulability condition of two non-preemptive strictly periodic tasks with communication constraints. In order to solve the scheduling problem, an

exact method is developed in Section 6.4. Furthermore, Section 6.5 proposes several heuristics in order to solve the same problem. Conclusion is stated in Section 6.6

6.1 Scheduling a strictly periodic independent tasks

This section presents the non-preemptive strictly periodic model and its characteristics. It presents the advantages of these tasks in presenting (modeling) critical real-time systems. Moreover, we recall the necessary and sufficient condition of scheduling two independent non-preemptive strictly periodic tasks on the same processor. Finally, we show that there exists a symmetry between the tasks starting dates which verify the schedulability condition.

6.1.1 Strictly periodic tasks

Strictly periodic non-preemptive tasks model is a particular case of the periodic tasks model. In this particular class, the task successive executions admit periodic starting dates. Moreover, the task execution cannot be interrupted by an other execution with higher priority, since the tasks are non-preemptive. The characteristics of non-preemptive strictly periodic tasks model are:

- the gap between the release dates of two successive executions of the same task (t_i) is equal to its period:

$$r_i^k - r_i^{k-1} = T_i, \quad \forall k \in \mathbb{N}^*.$$

- the gap between the starting dates of two successive executions of the same task (t_i) is equal to its period:

$$s_i^k - s_i^{k-1} = T_i, \quad \forall k \in \mathbb{N}^*.$$

- the gap between the starting date and the release date of each task execution (t_i^k) is equal to:

$$s_i^k - r_i^k = d, \quad \forall k \in \mathbb{N}^*.$$

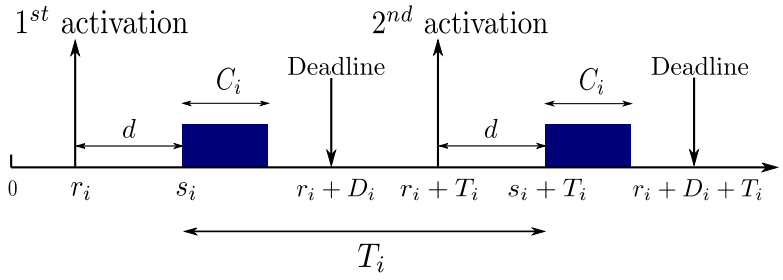


Fig. 6.1 Non-preemptive strictly periodic task.

Let $t_i = (r_i, C_i, D_i, T_i)$ be a non-preemptive strictly periodic task. Figure 6.1 illustrates the task executions. The task starts its first execution at s_i and ends it at $s_i + C_i$. The second execution begins at $s_i + T_i$ and ends at $s_i + T_i + C_i$. Furthermore, the gap between the task release date and its starting date in both executions is constant.

In critical real-time systems such as the control/command applications, the tasks which represent the system input and output should not have jitter. In order to avoid this jitter on the inputs receiving the data from sensors, input tasks must be strictly periodic. On the other hand, to avoid jitter on the outputs which transfer the data to actuators, output tasks must be non-preemptive.

In addition, for the majority of embedded systems, preemption is too expensive in time and space due to the context changing cost. This later is not controlled or not taken into account. For these reasons we focus our study on non-preemptive scheduling as long as we do not fully control the cost of preemption.

In the preemptive periodic model proposed by Liu and Layland [85], the gap between the start date of execution and the activation date may vary. However, this gap is constant in the strictly periodic non-preemptive tasks model. In fact, there is no jitter in the model [29].

Non-preemptive strictly periodic tasks are usually adopted in practical real-time applications when continual sampling and processing data are required. We find these types of applications in several domains such as avionics [37, 1] and process control [72, 114].

In the sequel of this manuscript, we use the term “ strictly periodic task ” to refer to a non-preemptive strictly periodic task.

6.1.2 Schedulabilty analysis of strictly periodic independent tasks

Considering a set of strictly periodic independent tasks $\mathcal{T} = \{t_1, \ldots, t_n\}$. Each task $t_i \in \mathcal{T}$ is characterized by an activation period T_i, an execution time C_i and a starting date (of the first execution) s_i. Since there is no precedence constraints between the tasks executions, we assume without loss of generality that $s_i \in [0, T_i)$. We use the interval $E_i^{\nu_i}(s_i)$ to characterize the execution time units occupied by the ν_ith execution of t_i having a starting date equals to s_i. The first execution interval of t_i begins at s_i and ends before $s_i + C_i$, hence $E_i^1(s_i) = [s_i, s_i + C_i)$. According to the task strict periodicity, the ν_ith execution of t_i begins at $s_i + (\nu_i - 1) \cdot T_i$ and ends before $s_i + (\nu_i - 1) \cdot T_i + C_i$. More formally, the execution time units for each task can be written as follows:

$$\forall t_i \in \mathcal{T}, \forall \nu_i \geq 1,\ E_i^{\nu_i}(s_i) = [s_i + (\nu_i - 1) \cdot T_i, s_i + (\nu_i - 1) \cdot T_i + C_i)$$

Let $t_i = (s_i, C_i, T_i)$ and $t_j = (s_j, C_j, T_j)$ be two strictly periodic independent tasks. These tasks are schedulable on the same processor if and only if there is no overlapping between their executions. This means $\forall (\nu_i, \nu_j) \in \mathbb{N}^{*^2}, E_i^{\nu_i}(s_i) \cap E_j^{\nu_j}(s_j) = \emptyset$. Therefore, a set of strictly periodic independent tasks $\mathcal{T} = \{t_1, \ldots, t_n\}$ can be scheduled on the same processor if and only if each tasks couple (t_i, t_j) can be scheduled on the same processor. This condition can be written as follows:

$$\underset{i \neq j}{\forall (t_i, t_j) \forall (\nu_i, \nu_j) \in \mathbb{N}^{*^2}}, E_i^{\nu_i}(s_i) \cap E_j^{\nu_j}(s_j) = \emptyset. \tag{6.1}$$

Although the previous equation (6.1) is a necessary and sufficient condition, it cannot be directly used to solve the scheduling problem of two tasks. Indeed, equation (6.1) requires the computation of the time units intervals occupied by all the tasks executions. However, the number of the tasks execution can increase exponentially and this is unacceptable in practice. Theorem 6.1.1 provides a schedulability condition without computing the interval execution time units occupied by all the tasks executions. This theorem was first proposed by Korst et al. [80]. More recently, Al Sheikh at al. [1] and Chen et al. [26] proved that this theorem does not require that the start date of the first task has to be smaller than the start date of the second one. This is done following the fact that $(-a) \mod (b) = b - (a) \mod b$.

Theorem 6.1.1 *Two strictly periodic independent tasks $t_i = (s_i, C_i, T_i)$ and $t_j = (s_j, C_j, T_j)$ are schedulable on the same processor if and only if*

$$C_i \leq (s_j - s_i) \mod (g_{ij}) \leq g_{ij} - C_j, \tag{6.2}$$

where $g_{ij} = \gcd(T_i, T_j)$.

Corollary 6.1.1 results from theorem 6.1.1. It gives a necessary and sufficient condition for scheduling the tasks on the same processor when their starting dates can be freely assigned by the designer.

Corollary 6.1.1 *[80] The executions of two periodic independent tasks* $t_i = (C_i, T_i)$ *and* $t_j = (C_j, T_j)$ *can be scheduled on the same processor if and only if*

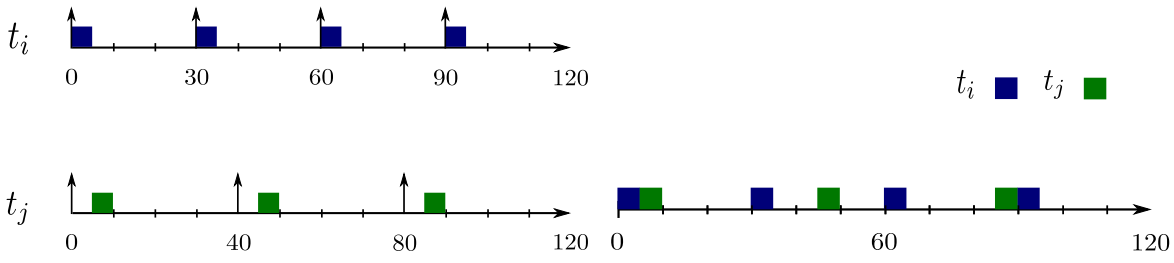
$$C_i + C_j \leq g_{ij}, \tag{6.3}$$

where $g_{ij} = \gcd(T_i, T_j)$.

Example 6.1.1 Let $t_i = (s_i, C_i, T_i)$ and $t_j = (s_j, C_j, T_j)$ be two independent strictly periodic tasks. We suppose that $T_i = 30$, $T_j = 40$ time units, $C_i = C_j = 5$ time units, and their starting dates are respectively equal to $s_i = 0$ and $s_j = 5$ time units. Figure 6.2a represents the tasks executions during their hyper period. We notice that there is no overlapping between the tasks executions (see Figure 6.2b). Therefore, t_i and t_j can be scheduled on the same processor. We can verify this result (without computing the tasks executions), by applying equation (6.2) as follows:

$$\begin{aligned} C_i \leq (s_j - s_i) &\mod (g_{ij}) \leq g_{ij} - C_j \\ 5 \leq \ (5-0) &\mod (10) \ \leq 10 - 5, \end{aligned}$$

where $g_{ij} = \gcd(30, 40) = 10$ time units.



(a) Tasks executions. (b) Non overlapping between executions.

Fig. 6.2 Scheduling two strictly periodic tasks $t_i = (0, 5, 30)$ and $t_j = (5, 5, 40)$ on the same processor.

Symmetry

Let $t_i = (s_i, C_i, T_i)$ and $t_j = (s_j, C_j, T_j)$ be two strictly periodic independent tasks. The difference between the tasks starting dates $(s_j - s_i)$ can be written as follows:

$$s_j - s_i = \delta_{ij} + k_{ij} \cdot g_{ij}, \tag{6.4}$$

where $\delta_{ij} = (s_j - s_i) \mod g_{ij}$ and $k_{ij} = \lfloor \frac{s_j - s_i}{g_{ij}} \rfloor$, with $k_{ij} \in \mathbb{Z}$.

On the other hand, the difference between the tasks starting dates $(s_i - s_j)$ can be written as follows:

$$s_i - s_j = \delta_{ji} + k_{ji} \cdot g_{ij}, \tag{6.5}$$

where $\delta_{ji} = (s_i - s_j) \mod g_{ij}$ and $k_{ji} = \lfloor \frac{s_i - s_j}{g_{ij}} \rfloor$, with $k_{ji} \in \mathbb{Z}$. According to the fact that $(-a) \mod (b) = b - (a) \mod (b)$ with $a, b > 0$, we can deduce that

$$\delta_{ji} = g_{ij} - \delta_{ij}.$$

Therefore, the equation (6.4) and (6.5) can also be written as follows:

$$\begin{aligned} s_j - s_i &= \delta_{ij} + k_{ij} \cdot g_{ij} \\ s_i - s_j &= g_{ij} - \delta_{ij} + k_{ji} \cdot g_{ij}. \end{aligned}$$

By summing these equalities, we obtain

$$k_{ij} \cdot g_{ij} + (k_{ji} + 1) \cdot g_{ij} = 0$$

Hence, we can deduce that

$$k_{ij} + k_{ji} = -1.$$

Finally, we can deduce a symmetry between δ_{ij} and δ_{ji} on one hand, and between k_{ij} and k_{ji} on the other hand. This implies that if there exists s_i and s_j which verify the equality (6.4) then these dates will also verify equality (6.5).

Let us consider the strictly periodic tasks $t_i = (0, 5, 30)$ and $t_j = (5, 5, 40)$ of our previous example. We notice that s_i and s_j verify equation (6.4) with $\delta_{ij} = 5$ and $k_{ij} = 0$. In addition, these dates verify equation (6.5) with $\delta_{ji} = g_{ij} - \delta_{ij} = 10 - 5 = 5$ and $k_{ji} = -1 - k_{ij} = -1$.

6.2 Synchronous Data Flow Graph schedule

SDFG is a static model which provides enough information in order to anticipate its behaviour. Therefore, it is possible to determine a scheduling for each task execution. Constraints that must be taken into consideration are the data dependencies that may exist between the tasks executions.

Let $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ be a SDFG. Given the set of tasks $\mathcal{A}$, solving a scheduling problem consists on finding the start dates for each task execution while respecting the associated constraints. More formally, a schedule is a function $s : \mathcal{T} \times \mathbb{N}^* \to \mathbb{R}_+$ which associates for each execution $t_i^{\nu_i}$ a starting date. We denote by $s_i^{\nu_i}$ the starting date of the ν_ith execution of t_i. Let $a = (t_i, t_j)$ be a SDFG buffer with its initial marking $M_0(a)$. There is a strong relationship between a schedule and the corresponding instantaneous marking of the buffer $M_t(a)$. This latter can be computed as follows:

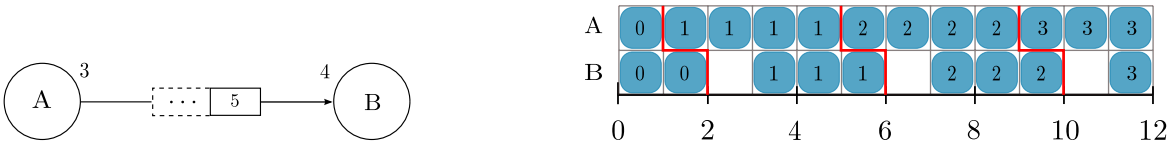
$$M_t(a) = M_0(a) + P_{t_i}^t \cdot z_i + C_{t_j}^t \cdot z_j,$$

where $P_{t_i}^t$ and $C_{t_j}^t$ correspond respectively to the number of completed executions of t_i and t_j at time t. A SDFG schedule is feasible if

$$\forall (t, a) \in \mathbb{R}_+^* \times \mathcal{A},\ M_t(a) \geq 0.$$

6.2.1 As Soon As Possible schedule

As Soon As Possible (ASAP in short) schedule is one of the most useful policy to analyse a SDFG. This scheduling algorithm consists in executing each task of the graph as soon as it has enough input data. ASAP schedule consists of two phases [28]: a transient phase followed by a periodic one where the tasks starting dates admit a periodic behaviour. Baccelli et al.[4] proved that the transient phase is bounded.



(a) A SDFG. (b) ASAP schedule

Fig. 6.3 Example of a SDFG and its ASAP schedule.

Figure 6.3b represents the ASAP schedule of the SDFG depicted in Figure 6.3a. In this example, the execution times of tasks A and B are fixed to 1 time unit. As

we can see in Figure 6.3b, the transition phase is limited to one execution of A and two executions of B (executions are marked by zeros). This phase is followed by a periodic phase where an execution pattern is repeated for each period (delimited by red lines). This pattern consists on executing task A four times and task B three times. The tasks' executions are marked according to the period number. For example, the executions marked by 1 correspond to the tasks' executions of the first period.

In general, the ASAP strategy does not guarantee a bounded marking. In fact, ensuring a bounded marking using ASAP strategy is restricted to some classes of strongly connected graphs. In addition, the drawback of the ASAP strategy for scheduling a SDFG, is the dependency from the SDFG repetition vector rather than its size ($|\mathcal{T}|$).

6.2.2 Periodic schedule

SDFGs are usually used to model applications that can be executed infinitely. Hence, the SDFG schedule must fix an infinite number of execution starting dates. In order to avoid this kind of issues, authors in [90, 12] were interested in the SDFG periodic schedules. They characterize each SDFG task t_i by an activation period ($\omega_i \in \mathbb{R}_+^*$) and a starting date of the first execution (s_i).

Definition 6.2.1 *Let $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ be a SDFG. A SDFG schedule is periodic if each task $t_i \in \mathcal{T}$ has a period ω_i such that*

$$\forall \nu_i \geq 1,\ s_i^{\nu_i} = s_i + (\nu_i - 1) \cdot \omega_i.$$

s_i is the starting date of the first execution of t_i. The successive executions of t_i are then repeated every ω_i time units.

Let $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ be a normalized SDFG. Considering a task $t_i \in \mathcal{A}$, let a be an output buffer of t_i. Every ω_i time units, z_i tokens are produced in the buffer a. Hence, on average, a token is produced every $\frac{\omega_i}{z_i}$ time units. This ratio is called average token flow time. Considering that all the tasks have the same average token flow time, Marchetti and Munier [90] prove that feasible periodic schedules satisfy linear inequalities. They defined a condition on the starting dates of the tasks first executions in order to fulfil the precedence constraints induced by the buffers.

Theorem 6.2.1 *[90] Let $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ be a normalized SDFG. For any periodic schedule, there exists a strictly positive rational $K \in \mathbb{Q}_+^*$ such that, for each task $t_i \in \mathcal{T}$, $\frac{\omega_i}{z_i} = K$.*

In addition, the precedence constraints induced by a buffer $a = (t_i, t_j)$ *are fulfilled by the periodic schedule if and only if*

$$s_j - s_i \geq C_i + K \cdot (z_j - M_0(a) - gcd_a). \tag{6.6}$$

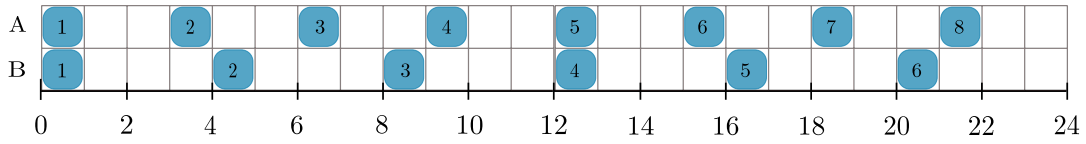


Fig. 6.4 A feasible periodic schedule of the SDFG depicted in Figure 6.3a.

Figure 6.4 illustrates a feasible periodic schedule of the SDFG presented in Figure 6.3a. The tasks periods are respectively equal to $\omega_A = 3$, $\omega_B = 4$ and their starting dates are null ($s_A = s_B = 0$). We notice that the precedence constraints induced by the buffer are fulfilled, since the tasks starting dates verify equation (6.6) with $C_A = 1$ and $K = \frac{\omega_A}{z_A} = \frac{\omega_B}{z_B} = 1$.

Unlike the ASAP strategy, the periodic schedule ensures a bounded marking for any consistent SDFG. In addition, a periodic schedule can be computed in polynomial time using the linear programming.

6.3 Schedulabilty analysis of two strictly periodic communicating tasks

Based on our communication model defined in Section 5.1 of Chapter 5, we prove that scheduling two strictly periodic communicating tasks is equivalent to schedule two independent tasks. These latter can only be executed between their release dates and their deadlines. In addition, we give a sufficient condition for scheduling two periodic communicating tasks on a single processor.

Lemma 6.3.1 *Let* $t_i = (r_i, C_i, D_i, T_i)$ *and* $t_j = (r_j, C_j, D_j, T_j)$ *be two strictly periodic communicating tasks. Let* $a = (t_i, t_j)$ *be the RTSDFG buffer that models the communications between the tasks executions. Let* s_i *and* s_j *be respectively the starting dates of* t_i^1 *and* t_j^1*. Communications constraints represented by the buffer* $a = (t_i, t_j)$ *are fulfilled, if*

$$s_i \in [r_i, r_i + D_i - C_i] \text{ and } s_j \in [r_j, r_j + D_j - C_j].$$

Proof: Let $a = (t_i, t_j)$ be the RTSDFG buffer that models the set of communication constraints between the strictly periodic tasks $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$. Let $s_i^{\nu_i}$ and $s_j^{\nu_j}$ be respectively the starting dates of $t_i^{\nu_i}$ and $t_j^{\nu_j}$. According to corollary 5.1.1, communication constraints induced by the buffer a are fulfilled, if

$$\forall \nu_i \geq 1,\ s_i^{\nu_i} \in [r_i + (\nu_i - 1) \cdot T_i,\ r_i + D_i - C_i + (\nu_i - 1) \cdot T_i] \text{ and,}$$
$$\forall \nu_j \geq 1,\ s_j^{\nu_j} \in [r_j + (\nu_j - 1) \cdot T_j,\ r_j + D_j - C_j + (\nu_j - 1) \cdot T_j].$$

On the other hand, according to the task strict periodicity, the execution starting dates $s_i^{\nu_i}$ and $s_j^{\nu_j}$ can be computed in function of the tasks first executions (s_i and s_j) as following:

$$s_i^{\nu_i} = s_i + (\nu_i - 1) \cdot T_i$$
$$s_i^{\nu_j} = s_j + (\nu_j - 1) \cdot T_j.$$

Consequently, it is sufficient that $s_i \in [r_i, r_i + D_i - C_i]$ and $s_j \in [r_j, r_j + D_j - C_j]$ (for $\nu_i = \nu_j = 1$) to ensure that all communications constraints induced by the buffer a are satisfied. □

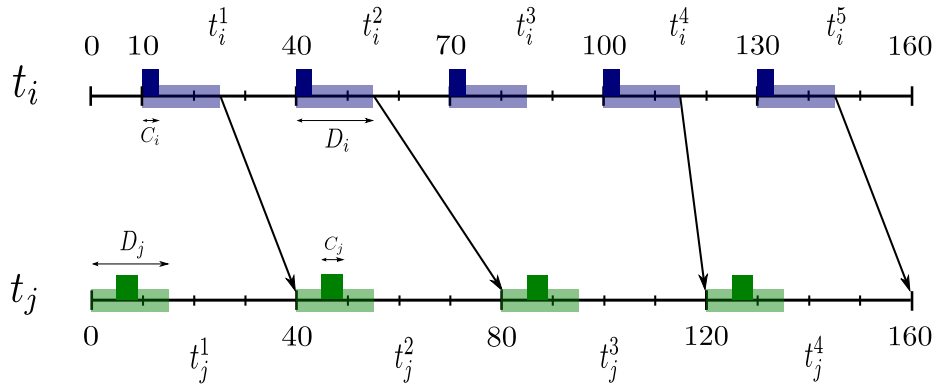


Fig. 6.5 Communications constraints between the executions of two strictly periodic tasks $t_i = (10, 3, 15, 30)$ and $t_j = (0, 4, 15, 40)$.

Example 6.3.1 Let $t_i = (10, 3, 15, 30)$ and $t_j = (0, 4, 15, 40)$ be two strictly periodic communicating tasks. Figure 6.5 represents the communications between the tasks executions. We suppose that the tasks start their executions respectively at $s_i = 10$

and $s_j = 5$ time units. Communication constraints are fulfilled, since the first execution of each task is accomplished between its release date and its deadline ($s_i \in [10, 22]$ and $s_j \in [0, 11]$).

Theorem 6.3.1 *Let $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$ be two strictly periodic communicating tasks. Let s_i and s_j be respectively the starting dates of t_i^1 and t_j^1 such that $s_i \in [r_i, r_i + D_i - C_i]$ and $s_j \in [r_j, r_j + D_j - C_j]$. These tasks can be scheduled on the same processor if and only if*

$$C_i \leq (s_j - s_i) \mod g_{ij} \leq g_{ij} - C_j,$$

where $g_{ij} = \gcd(T_i, T_j)$.

Proof: Considering $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$ two strictly periodic communicating tasks. Let $a = (t_i, t_j)$ be the RTSDFG buffer that models the communication constraints between the tasks executions. Let s_i and s_j be respectively the starting dates of t_i^1 and t_j^1 such that $s_i \in [r_i, r_i + D_i - C_i]$ and $s_j \in [r_j, r_j + D_j - C_j]$. Consequently, according to lemma 6.3.1, communications constraints induced by the RTSDFG buffer $a = (t_i, t_j)$ are fulfilled. Hence, we can deduce that tasks $t_i = (s_i, C_i, T_i)$ and $t_j = (s_j, C_j, T_j)$ can be considered as independent tasks, since the starting dates of their first executions s_i and s_j satisfy the communications constraints.

On the other hand, two strictly periodic independent tasks are scheduled on the same processor if and only if there is no overlapping between their executions. According to theorem 6.1.1, these tasks can be executed on the same processor if and only if the starting dates of their first executions satisfy the following equation:

$$C_i \leq (s_j - s_i) \mod g_{ij} \leq g_{ij} - C_j.$$

□

Let us consider the strictly periodic communicating tasks $t_i = (10, 3, 15, 30)$ and $t_j = (0, 4, 15, 40)$ of the example in Figure 6.5. Tasks are executed between their release dates and their deadlines. Therefore, communication constraints between their executions are fulfilled. Accordingly, these tasks can be considered as independent ones. Moreover, we notice that the tasks starting dates $s_i = 10$ and $s_j = 5$ verify the condition of scheduling two independent tasks on the same processor (equation (6.2)):

$$\begin{aligned} C_i \leq (s_j - s_i) &\mod g_{ij} \leq g_{ij} - C_j \\ 3 \leq (5 - 10) &\mod 10 \leq 10 - 4. \end{aligned}$$

Hence, we can deduce that t_i and t_j can be scheduled on the same processor while respecting the communication constraints between their executions (see Figure 6.6).

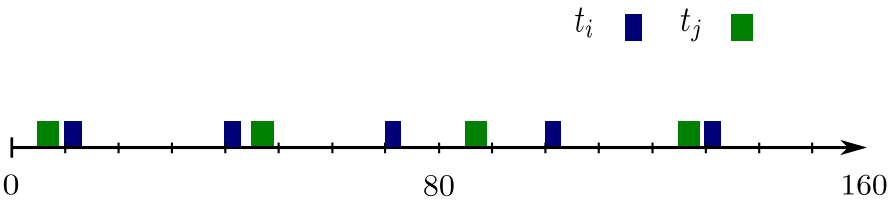


Fig. 6.6 Scheduling on the same processor the strictly periodic communicating tasks $t_i = (10,3,15,30)$ and $t_j = (0,4,15,40)$ of the example in Figure 6.5.

Schedulability condition of two strictly periodic communicating tasks

This subsection presents a sufficient condition for scheduling two strictly periodic tasks on a the same processor. This condition takes into consideration the communication constraints between the tasks executions and the available resource of a single processor. In other words, given two strictly periodic tasks with their starting dates, the sufficient condition allows to verify whether these tasks can be scheduled on a the same processor or not.

Corollary 6.3.1 *Let $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_i, T_j)$ be two strictly periodic communicating tasks. Let $a = (t_i, t_j)$ be a RTSDFG buffer that models the set of communications constraints between the tasks executions. Let s_i and s_j be respectively the starting dates of t_i^1 and t_j^1. These tasks can be scheduled to the same processor if:*

$$\max\{C_i, \mathcal{M}\} \leq (s_j - s_i) \mod g_{ij} \leq \min\{g_{ij} - C_j, \mathcal{M}^{'}\} \tag{6.7}$$

with $\mathcal{M} = r_j - r_i - D_i + C_i - k_{ij} \cdot g_{ij}$ and $\mathcal{M}^{'} = r_j - r_i + D_j - C_j - k_{ij} \cdot g_{ij}$

Proof: Let $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$ be two strictly periodic communicating tasks. Let $a = (t_i, t_j)$ be a RTSDFG buffer that models the set of communications constraints between the tasks' executions. Let s_i and s_j be respectively the starting dates of t_i^1 and t_j^1. The gap between the tasks starting dates can be written as follows:

$$s_j - s_i = \delta_{ij} + k_{ij} \cdot g_{ij}$$
$$\delta_{ij} = s_j - s_i - k_{ij} \cdot g_{ij},$$

where $\delta_{ij} = (s_j - s_i) \mod g_{ij}$ and $k_{ij} = \lfloor \frac{s_j - s_i}{g_{ij}} \rfloor$. According to theorem 6.1.1, two strictly periodic independent tasks can be scheduled on the same processor if

$$\begin{aligned} C_i \leq \quad & \delta_{ij} \quad \leq g_{ij} - C_j \\ C_i \leq (s_j - s_i) - k_{ij} \cdot g_{ij} & \leq g_{ij} - C_j. \end{aligned} \tag{6.8}$$

On the other hand, there exists a set of communication constraints between the executions of t_i and those of t_j. According to lemma 6.3.1, communication constraints between the tasks' executions are fulfilled if the starting dates of their first executions belong to the following intervals:

$$\begin{aligned} r_i \leq s_i \leq r_i + D_i - C_i \\ r_j \leq s_j \leq r_j + D_j - C_j. \end{aligned}$$

Hence, the gap between s_i and s_j can be written as follows:

$$r_j - r_i - D_i + C_i \leq s_j - s_i \leq r_j - r_i + D_j - C_j$$

Therefore, the communication constraints between the tasks' executions are fulfilled if $\delta_{ij} = (s_j - s_i) - k_{ij} \cdot g_{ij}$ satisfy the following inequality:

$$r_j - r_i - D_i + C_i - k_{ij} \cdot g_{ij} \leq (s_j - s_i) - k_{ij} \cdot g_{ij} \leq r_j - r_i + D_j - C_j - k_{ij} \cdot g_{ij} \tag{6.9}$$

By combining equations (6.8) and (6.9), we can deduce that two strictly periodic communicating tasks can be scheduled on the same processor if

$$\max\{C_i, r_j - r_i - D_i + C_i - k_{ij} \cdot g_{ij}\} \leq \delta_{ij} \leq \min\{g_{ij} - C_j, r_j - r_i + D_j - C_j - k_{ij} \cdot g_{ij}\}$$

□

Considering the strictly periodic communicating tasks $t_i = (10, 3, 15, 30)$ and $t_j = (0, 4, 15, 40)$. We assume that the tasks starting dates are respectively equal to $s_i = 10$ and $s_j = 5$. Scheduling these tasks on the same processor can be verified by applying equation (6.7) as follows:

$$\begin{aligned} \max\{C_i, \mathcal{M}\} \leq (s_j - s_i) \quad & \mod g_{ij} \leq \min\{g_{ij} - C_j, \mathcal{M}'\} \\ \max\{3, -12\} \leq (5 - 10) \quad & \mod 10 \leq \min\{10 - 4, 11\}, \end{aligned}$$

with $\mathcal{M} = 0 - 10 - 15 + 3 - (-1) \cdot 10 = -12$ and $\mathcal{M}' = 0 - 10 + 15 - 4 - (-1) \cdot 10 = 11$.

6.4 Optimal algorithm: Integer linear programming formulation

A linear programming (LP in short) problem consists on maximizing or minimizing a linear function, subject to a finite number of linear constraints. These latter may be equalities or inequalities. The problem feasible region is a convex polytope which is defined by linear inequalities. Its objective function is a real-valued affine function defined on this polytope. A linear programming algorithm consists on searching through the polytope vertices in order to find a point (on this polytope) where the objective function has the smallest or the largest value if such a point exists. The linear programming problem was first introduced by Kantorovich [68] who also proposed a method for solving it. The well know Simplex method, which is an important resolution method dedicated to solve linear programs, was introduced by Dantzig [31]. The LP problem can be solved using the Simplex method if there are no restrictions on the variables types.

An integer linear programming (ILP in short) is a linear program with additional constraint which requires that all the variables are restricted to integer values. When the integer restriction is applied only to a subset of the variables, the problem is defined as Mixed Integer Linear Programming (MILP in short). Several exact algorithms are dedicated to solve these problems such as branch-and-bound [82], branch-and-cut [63], branch-and-price [105] and cutting-plane [88]. Moreover, these linear programming algorithms are known to be exact.

In this thesis, we need to answer the following question:

For a given set of strictly periodic tasks, these latter can be scheduled on the same processor, assuming that the tasks communicate with each other and can not be pre-empted?

In this section, an exact algorithm is suggested for the mono-processor scheduling problem. This latter consists of scheduling a set of strictly periodic communicating tasks on a single processor. Our approach used the mixed integer linear programming formulation in order to build an exact framework. Our MILP formulation seeks a feasible solution that satisfies the constraints induced by the communications and available resource of a single processor without having any objective function.

Consider $\mathcal{N} = \{t_1, \ldots, t_n\}$ a set of strictly periodic communicating tasks. Let $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ be the RTSDFG which models all the communications constraints between

the tasks executions. According to lemma 6.3.1, the set of communication constraints is fulfilled if the starting date of the first execution (s_i) of each task belongs to the following interval:

$$\forall t_i \in \mathcal{T},\ s_i \in [r_i, r_i + D_i - C_i].$$

In our study, we distinguish two cases: *fixed* and *flexible* intervals.

6.4.1 *Fixed* intervals

In this subsection, we presents the MILP formulation of the *fixed* intervals case, where each task $t_i = (r_i, C_i, D_i, T_i)$ can only start its first execution during the interval $[r_i, r_i + D_i - C_i]$. Solving the MILP seeks to find a feasible solution if such one exists. A feasible solution is composed of a set of starting dates which verify simultaneously the communication and the resource constraints. Prior to presentation of the general model, we present a list of constants and variables used in the MILP formulation.

Constants

- r_i : Release date of the task first execution (t_i^1).
- C_i : Execution time of task t_i.
- D_i : Relative deadline of task t_i.

Variables

- s_i : Starting date of the task's first execution (t_i^1).
- $\delta_{i,j}$: For each pair of tasks (t_i, t_j), δ_{ij} corresponds to the time interval which is allowed to occupy without interfering with each other.
- $k_{i,j}$: For each pair of tasks (t_i, t_j), $k_{i,j}$ corresponds to the number of intervals of length g_{ij} separating the starting dates of the tasks' first executions

Constraints

- Communication constraints: According to lemma 6.3.1, communication constraints between the tasks executions are fulfilled if the first starting date of each task execution belongs to the following interval

$$\forall t_i \in \mathcal{T},\ r_i \leq s_i \leq r_i + D_i - C_i$$

- Resource constraints: A set of strictly periodic communicating tasks $\mathcal{T} = \{t_1, \ldots, t_n\}$ can be scheduled on the same processor if and only if each couple of tasks (t_i, t_j) can be scheduled on this processor. According to theorem 6.3.1, two communicating tasks are executed on the same processor if and only if equation 6.2 is fulfilled,

$$C_i \leq (s_j - s_i) \mod g_{ij} \leq g_{ij} - C_j.$$

In equation 6.2, mod is not a linear operation. In MILP formulation, this equation must be substitute by the following one:

$$s_j - s_i = \delta_{ij} + k_{ij} \cdot g_{ij}$$

where $C_i \leq \delta_{ij} \leq g_{ij} - C_j$ and $k_{ij} = \lfloor \frac{s_j - s_i}{g_{ij}} \rfloor$.

Moreover, and according to the symmetry presented in Section 6.1, we can halve the constraints number. Therefore, the schedulability condition is only verified for each couple (t_i, t_j) with $i < j$. More formally,

$$\forall \underset{i<j}{(t_i, t_j) \in \mathcal{T}^2}, \; s_j - s_i = \delta_{ij} + k_{ij} \cdot g_{ij}$$

The following MILP gathers all the scheduling problem constraints of the *fixed* intervals case, as follows:

subject to

$$\begin{array}{ll}
r_i \leq s_i \leq r_i + D_i - C_i, & \forall t_i \in \mathcal{T} \\
s_j - s_i = \delta_{i,j} + k_{i,j} \cdot g_{i,j}, & \forall \underset{i<j}{(t_i, t_j) \in \mathcal{T}^2} \\
C_i \leq \delta_{i,j} \leq g_{i,j} - C_j, & \forall \underset{i<j}{(t_i, t_j) \in \mathcal{T}^2} \\
s_i \in \mathbb{R}_+ & \forall t_i \in \mathcal{T} \\
k_{i,j} \in \mathbb{Z}, \; \delta_{ij} \in \mathbb{R}_+ & \forall \underset{i<j}{(t_i, t_j) \in \mathcal{T}^2}
\end{array}$$

Example 6.4.1 Consider the set of strictly periodic communicating tasks $\mathcal{N} = \{t_1, t_2, t_3\}$. Let $\mathcal{E} = \{(t_1, t_2), (t_2, t_3), (t_3, t_1)\}$ be the set of communication relationships between tasks. Table 6.1 represents the tasks parameters. Figure 6.7 depicts the RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ that models the communications between the tasks executions.

t_i	r_i	C_i	D_i	T_i
t_1	90	10	30	120
t_2	150	30	140	240
t_3	30	20	30	60

Table 6.1 Strictly periodic tasks parameters

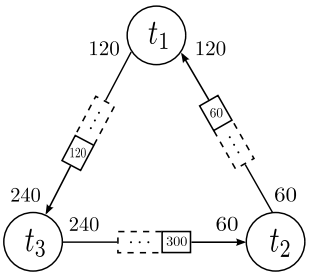


Fig. 6.7 RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$.

Our purpose is to schedule these tasks on the same processor while respecting the communication constraints between their executions. We reach our target by solving the MILP formulation that seeks to find a feasible solution, if such a solution exists. Figure 6.8 represents a feasible (schedule) solution of this problem where the tasks starting dates are respectively equal to $s_1 = 110$, $s_2 = 180$ and $s_3 = 30$ time units. We notice that all the tasks are executed between their release dates and their deadlines. Therefore, the communication constraints between the tasks' executions are fulfilled. For example, t_3^1 starts at 30 time units and ends at 50 time units. Data produced during its execution are available at 60 time units. These data are consumed by t_1^1 at 90 time units. In turn, t_1^1 starts its execution at 110 time units and ends it at 120 time units. In addition, we notice that there is no overlapping between the tasks executions. Hence, we can deduce that the resource constraints are also respected.

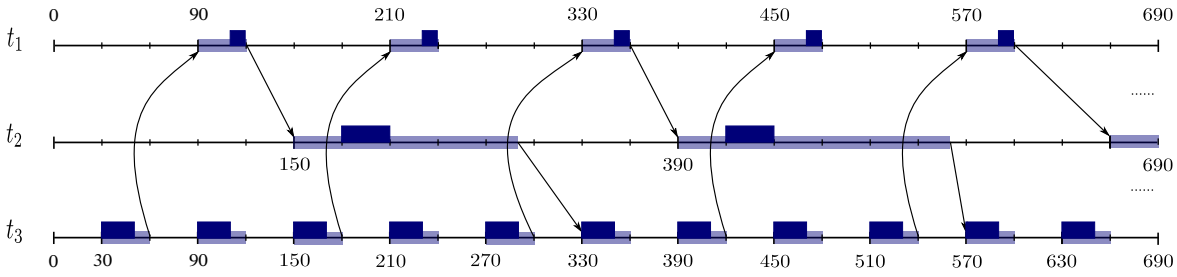


Fig. 6.8 A feasible Schedule of a strictly periodic system with communication constraints.

Infeasible solution

Let us modify the previous example by increasing the execution time of task t_1 from 10 time units to 20 time units. Let $s_1 = 100$, $s_2 = 180$ and $s_3 = 30$ be respectively the tasks starting dates. Figure 6.9 illustrates the communication and execution model of these tasks. Communication constraints are fulfilled since each task is executed between its release date and its deadline. However, we notice that t_1 and t_3 can not

admit starting dates that verify simultaneously the communication and the resource constraints. In fact, there is an overlapping between the executions of t_1 and that of t_3 if these tasks are executed between their release dates and their deadlines. For example, the first execution of t_1 begins at 100 and ends at 120 time units. In addition, the second execution of t_3 begins at 90 time units and it is accomplished after the beginning of t_1^1 (at 110 time units).

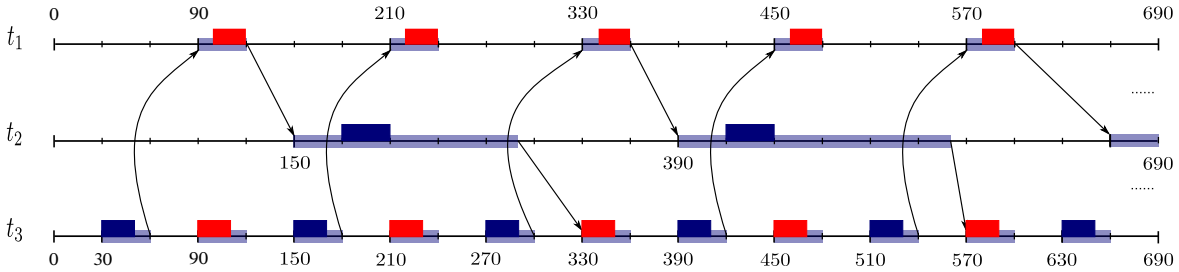


Fig. 6.9 Example of an infeasible solution where the resource constraints are not fulfilled. There is an overlapping between the executions of t_1 and that of t_3.

6.4.2 *Flexible* intervals

In this subsection, we introduce the MILP formulation of the *flexible* intervals case. Unlike the *fixed* intervals case, the beginning of the first execution of each task $t_i = (r_i, C_i, D_i, T_i)$ is no more limited (restricted) by the interval $[r_i, r_i + D_i - C_i]$. Tasks can start their executions in new execution intervals where the communications constraints between their executions are always fulfilled. As the *fixed* intervals case, solving the MILP seeks to find a feasible solution if such one exists. Prior to the definition of the general model, we present a list of constants and variables used in the MILP formulation.

Constants

- r_i : Release date of the task's first execution (t_i^1).
- C_i : Execution time of task t_i.
- D_i : Relative deadline of task t_i.

Variables

- r_i^* : Starting date of the new execution interval ($[r_i^*, r_i^* + D_i]$).

- s_i : Starting date of the task first execution (t_i^1).
- $\delta_{i,j}$: For each pair of tasks (t_i, t_j), δ_{ij} corresponds to the time interval which is allowed to occupy without interfering with each other.
- $k_{i,j}$: For each pair of tasks (t_i, t_j), $k_{i,j}$ corresponds to the number of intervals of length g_{ij} separating the starting dates of the tasks' first executions

Constraints

- Communication constraints: According to theorem 5.1.1 in section 5.1 of chapter 5, the set of communication constraints between two periodic tasks $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$ is modeled by a RTSDFG buffer $a = (t_i, t_j)$. Its production and consumption rates are respectively equal to $z_i = T_i$ and $z_j = T_j$ and its initial marking is equal to $M_0(a)$. Moreover, and according to theorem 6.2.1, the precedence constraints induced by the buffer $a = (t_i, t_j)$ are fulfilled by a periodic schedule if and only if

$$s_j - s_i \geq C_i + K \cdot (z_j - M_0(a) - gcd_a),$$

where $K = \frac{T_i}{z_i} = \frac{T_j}{z_j} \in \mathbb{Q}_+^*$.

In our case, K is equal to $\frac{T_i}{z_i} = \frac{T_j}{z_j} = 1$. The set of communications modeled by the RTSDFG buffer is build according to the task's release dates (r_i) and their relative deadlines (D_i). In fact, a RTSDFG's task corresponds to the interval ($[r_i, r_i + D_i]$) during which the strictly periodic t_i task is executed. Therefore, the starting date of the RTSDFG task can be considered as the starting date of the new execution interval (r_i^*). Accordingly, precedence constraints between the new execution intervals are fulfilled if and only if

$$\forall a = (t_i, t_j) \in \mathcal{A},\ r_j^* - r_i^* \geq D_i + (T_j - M_0(a) - gcd_a).$$

Then, the new execution interval must begin after the task release date,

$$\forall t_i \in \mathcal{T},\ r_i^* \geq r_i.$$

Finally, and according to lemma 6.3.1, communications constraints between the tasks executions are fulfilled if the starting dates of their first execution belongs

to the new execution intervals,

$$\forall t_i \in \mathcal{T},\ r_i^* \leq s_i \leq r_i^* + D_i - C_i.$$

- Resource constraints: As in the *fixed* intervals case, the set of strictly periodic communicating tasks can be scheduled on the same processor if and only if:

$$\forall \underset{i<j}{(t_i,t_j) \in \mathcal{T}^2},\ s_j - s_i = \delta_{ij} + k_{ij} \cdot g_{ij}$$

where $C_i \leq \delta_{ij} \leq g_{ij} - C_j$ and $k_{ij} = \lfloor \frac{s_j - s_i}{g_{ij}} \rfloor$.

The following MILP formulation gathers all the scheduling problem constraints of the *flexible* intervals case:

subject to

$$\begin{array}{ll}
r_j^* - r_i^* \geq D_i + (T_j - M_0(a) - gcd_a), & \forall a = (t_i,t_j) \in \mathcal{A} \\
r_i^* \geq r_i, & \forall t_i \in \mathcal{T} \\
r_i^* \leq s_i \leq r_i^* + D_i - C_i, & \forall t_i \in \mathcal{T} \\
s_j - s_i = \delta_{i,j} + k_{i,j} \cdot g_{i,j}, & \forall \underset{i<j}{(t_i,t_j) \in \mathcal{T}^2} \\
C_i \leq \delta_{i,j} \leq g_{i,j} - C_j, & \forall \underset{i<j}{(t_i,t_j) \in \mathcal{T}^2} \\
(s_i, r_i^*) \in \mathbb{R}_+^2 & \forall t_i \in \mathcal{T} \\
k_{i,j} \in \mathbb{Z},\ \delta_{ij} \in \mathbb{R}_+ & \forall \underset{i<j}{(t_i,t_j) \in \mathcal{T}^2}
\end{array}$$

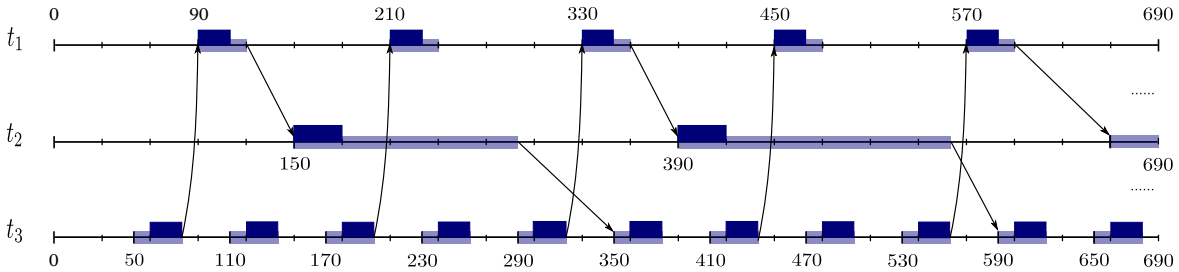


Fig. 6.10 A feasible schedule of the example depicted in Figure 6.9. Communication constraints are respected and there is no overlapping between the tasks executions.

Example 6.4.2 Let us reconsider the strictly periodic system depicted in figure 6.9. Unlike the *fixed* intervals case, this system admits a feasible schedule. Figure 6.10

illustrates a feasible solution where the tasks starting dates are respectively equal to $s_1 = 90$, $s_2 = 150$ and $s_3 = 60$. In addition, the starting date of the new execution intervals are respectively equal to $r_1^* = 90$, $r_2^* = 150$ and $r_3^* = 50$. We notice that the communication constraints are fulfilled, since the tasks are executed during the interval $[r_i^*, r_i^* + D_i]$. For example, t_3^1 begins its execution at 60 time units and it is accomplished at 80 time units. Data produced during its execution are consumed by t_1^1 at 90 time units. Moreover, we notice that there is no overlapping between the tasks executions. For instance, t_1^1 starts at 90 time units and ends at 110 time units. On the other hand, t_3^1 is accomplished at 80 time units and t_3^2 starts at 120 time units.

In order to justify the MILP formulation of the *flexible* intervals case, more precisely the addition of the variable r_i^* (starting date of the new execution interval), we consider the following example:

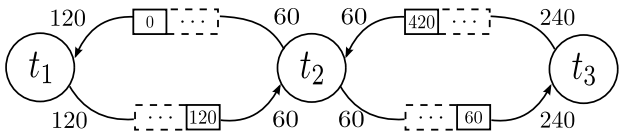


Fig. 6.11 RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ modeling the communication constraints between the executions of $t_1 = (110, 10, 25, 120)$, $t_2 = (15, 15, 30, 60)$ and $t_3 = (190, 45, 200, 240)$.

Example 6.4.3 Let $t_1 = (110, 10, 25, 120)$, $t_2 = (15, 15, 30, 60)$ and $t_3 = (190, 45, 200, 240)$ be three strictly periodic communicating tasks. Figure 6.11 depicts the RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ that models the set of communication constraints between the tasks' executions. According to our communication model, data are only available at the task's deadline and they can only be consumed at the task's release date. Figure 6.12 illustrates the communication scheme that should be respected in order to fulfil the communication constraints between the tasks' executions. For example, execution t_2^3 must communicate with t_3^1 which in turn must communicate with t_2^8. Solving the MILP formulation of the *flexible* intervals case provides us a feasible schedule. Figure 6.13 illustrates this feasible solution where the tasks' starting dates $s_1 = 130$, $s_2 = 55$ and $s_3 = 190$ time units. In addition, the interval starting dates are $r_1^* = 130$, $r_2^* = 40$ and $r_3^* = 190$ time units.

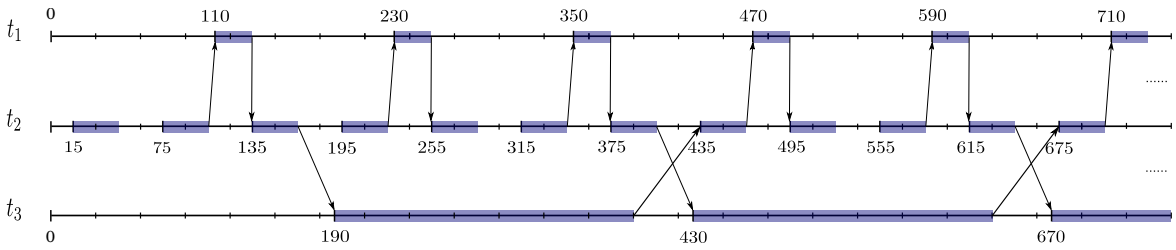


Fig. 6.12 Communication scheme between the executions of$t_1 = (110, 10, 25, 120)$, $t_2 = (15, 15, 30, 60)$ and $t_3 = (190, 45, 200, 240)$.

In order to highlight the impact of the variable r_i^* to solve the mono-processor scheduling, we assume that the tasks' starting dates are equal to the intervals starting dates ($s_i = r_i^*, \quad \forall t_i \in \mathcal{T}$). This means that the tasks must start their executions at the beginning of their intervals. Let us reconsider the feasible solution depicted in Figure 6.13. The interval starting dates of tasks t_1 and t_3 remain unchanged since their executions start at the beginning of their intervals. However, the interval starting date of task t_2 will be modified so that it is equal to $s_2 = 55$ time units. By changing this value, we notice that the resource constraints remain respected. However, this is no longer the case for the communication constraints, since t_2^3 is not able to transfer the data to t_3^1. In fact, the deadline of t_2^3 is greater than the release date of t_3^1. On the other hand, let us decrease the starting date of t_2^1 from 55 to 40 time units so that t_2 is executed at the beginning of its interval. In this case, we notice that the communication constraints remain respected. However, this is no longer the case for the resource constraints, since there is an overlapping between t_2^4 and t_3^1. Therefore, we deduce that adding the starting date of the new execution interval (r_i^*) to the MILP formulation is essential to solve the mono-processor scheduling problem for the *flexible* interval case.

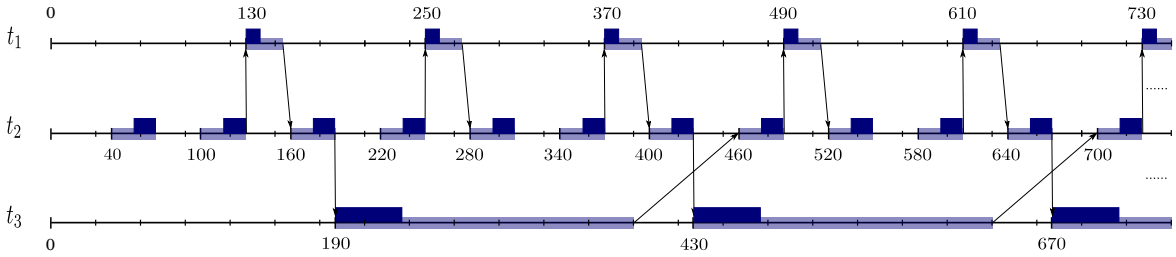


Fig. 6.13 Scheduling the strictly periodic communicating tasks $t_1 = (110, 10, 25, 120)$, $t_2 = (15, 15, 30, 60)$ and $t_3 = (190, 45, 200, 240)$ on the same processor.

6.5 Heuristics

Solving the MILP produces a schedule for a set of strictly periodic communicating tasks on a single processor. Numerically solving the MILP within a reasonable amount of time is only restricted to small instance. Recall that the scheduling problem in question is known to be NP-complete in the strong sense [79]. Consequently, we present in this section three heuristics which seek to find a feasible solution for our scheduling problem, such as linear programming relaxation. For the second and the third heuristic if no feasible solution is found, a partial solution is computed. This solution corresponds to a subset of tasks that can be executed on the same processor.

6.5.1 Linear programming relaxation

Linear programming relaxation is a method which consists on transforming a combinatorial or discrete problem to a continuous one. This method is used to obtain information on the initial discrete problem in order to build a solution. The linear programming relaxation may be solved using any standard linear programming technique (such as Simplex). Our first approach is based on the linear programming relaxation and it can be defined in the following steps:

- Relaxation of the integrality constrain: this relaxation allows variables to have non-integral values. The MILP of the *Flexible* intervals case can be " relaxed " as follows :

subject to

$$
\begin{array}{ll}
r_j^* - r_i^* \geq D_i + (T_j - M_0(a) - gcd_a), & \forall a = (t_i, t_j) \in \mathcal{A} \\
r_i^* \geq r_i, & \forall t_i \in \mathcal{T} \\
r_i^* \leq s_i \leq r_i^* + D_i - C_i, & \forall t_i \in \mathcal{T} \\
s_j - s_i = \delta_{i,j} + k_{i,j} \cdot g_{i,j}, & \forall (t_i, t_j) \in \mathcal{T}^2 \\
 & \quad i<j \\
C_i \leq \delta_{i,j} \leq g_{i,j} - C_j, & \forall (t_i, t_j) \in \mathcal{T}^2 \\
 & \quad i<j \\
(s_i, r_i^*) \in \mathbb{R}_+^2 & \forall t_i \in \mathcal{T} \\
k_{i,j} \in \mathbb{R},\ \delta_{ij} \in \mathbb{R}_+ & \forall (t_i, t_j) \in \mathcal{T}^2 \\
 & \quad i<j
\end{array}
$$

- Solving the resulting linear program in order to obtain a fractional optimal solution. We can notice that all the linear program constraints are potential

except the resource ones. For example, the precedence constraints induced by the RTSDFG buffers are potential constraints, since the gap between the starting date of the new execution intervals $(r_j^* - r_i^*)$ are greater than or equal to an integer constant. However, the equality $s_j - s_i = \delta_{i,j} + k_{i,j} \cdot g_{i,j}$ is not potential.

- For each couple of tasks $\underset{i<j}{(t_i, t_j)} \in \mathcal{T}^2$, we compute the integer value of k_{ij} which is equal to $\lfloor \frac{s_j - s_i}{g_{ij}} \rfloor$.
- We substitute each k_{ij} with its value in a new execution LP where k_{ij} is no longer considered as a variable. In order to verify the existence of a feasible (integer) solution to our scheduling problem, we solve the new LP. Note that finding a feasible solution for the new LP is equivalent to find a feasible solution for the original MILP.

Let us reconsider the example depicted in figure 6.8. Fractional solution obtained by solving the " relaxed " program is equal to:

$$\begin{array}{llll} s_1 = 140 & r_1^* = 120 & k_{12} = 0.9167 & \delta_{12} = 10, \\ s_2 = 260 & r_2^* = 150 & k_{13} = -0.8334 & \delta_{13} = 10, \\ s_3 = 100 & r_3^* = 90 & k_{23} = -3.1667 & \delta_{23} = 30. \end{array}$$

For each couple of tasks $\underset{i<j}{(t_i, t_j)} \in \mathcal{T}^2$, we compute the number of intervals of length g_{ij} separating the starting dates of the tasks' executions:

$$\begin{aligned} k_{12} &= \lfloor \frac{s_2 - s_1}{g_{12}} \rfloor = \lfloor \frac{260 - 140}{120} \rfloor = 1, \\ k_{13} &= \lfloor \frac{s_3 - s_1}{g_{13}} \rfloor = \lfloor \frac{100 - 140}{60} \rfloor = -1, \\ k_{23} &= \lfloor \frac{s_3 - s_2}{g_{23}} \rfloor = \lfloor \frac{100 - 260}{60} \rfloor = -3. \end{aligned}$$

By solving the new LP where each k_{ij} is substituted by its integer value, we obtain the following solution:

$$\begin{array}{lll} s_1 = 90 & r_1^* = 90 & \delta_{12} = 10, \\ s_2 = 220 & r_2^* = 150 & \delta_{13} = 40, \\ s_3 = 70 & r_3^* = 60 & \delta_{23} = 30. \end{array}$$

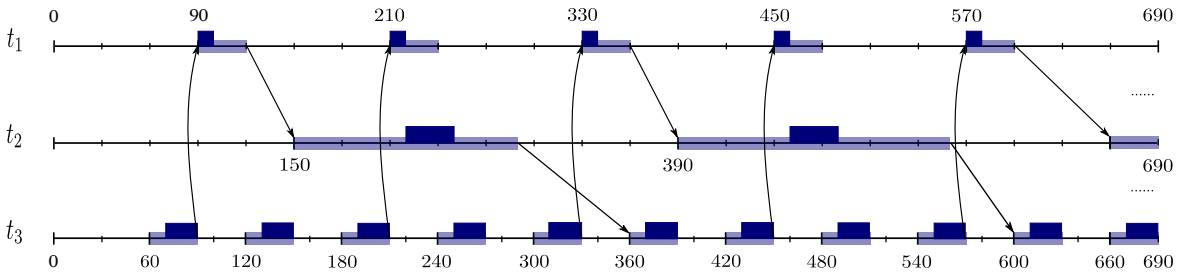


Fig. 6.14 A feasible solution for the scheduling problem is obtained by solving the new LP with k_{12}, k_{13} and k_{23} respectively equal to $1, -1$ and -3.

Figure 6.14 illustrates the feasible solution obtained by solving the new LP. We notice that there is no overlapping between the tasks executions and the communication constraints between them are fulfilled.

6.5.2 *Simple* heuristic

In this subsection, we describe in detail our *simple* heuristic for scheduling strictly periodic communicating tasks. It is based on the schedulability condition of two strictly periodic communicating tasks, which is presented previously in Section 6.3. In our approach, we use a greedy algorithm without back-tracking (i.e. if a decision is made at some stage, it is never questioned during the following stages). The effectiveness of any greedy algorithm is predicated on the decisions choice. In the *simple* heuristic, decisions are based on two criteria: the interval length where each task can start its execution ($[r_i, r_i + D_i - C_i]$) and the task period (T_i).

Our algorithm takes as input a sorted list of tasks (the sort description is represented subsequently). In addition, this algorithm returns a feasible solution (if it is able to find such a solution). Otherwise, it returns a partial solution that corresponds to a subset of tasks with their starting dates which can be executed on the same processor.

We sort the set of tasks in an ascending order according to the gap between the tasks relative deadlines and their execution times ($D_i - C_i$). Moreover, if two tasks have the same gap between their deadlines and their execution times, the task with the smallest period has a higher priority. In other words, the task with the smallest period is placed prior to the other task in the sorted list. More formally, we denote by $\mathcal{LT}$ the sorted list of tasks such that

$\mathcal{LT} = \{t_i : \forall t_i \in \mathcal{T},\ (D_i - C_i) \leq (D_{i+1} - C_{i+1})$. If $(D_i - C_i) = (D_{i+1} - C_{i+1})$ then the task with the smallest period has a higher priority$\}$.

Algorithm 4 describes the *simple* heuristic as follows:

- First, we initialize an empty list $\mathcal{S}_{olution}$. We insert in this latter the first task of $\mathcal{LT}$ while setting its starting date to its release date $s_1 = r_1$ (line 3). In addition, we remove this task from $\mathcal{LT}$ (line 4). $\mathcal{S}_{olution}$ corresponds to the set of scheduled tasks in the partial or global scheduling.
- Then, we verify for each candidate task t_j in $\mathcal{LT}$ if there exists a starting date $s_j \in [r_j, r_j + D_j - C_j]$ which fulfils the schedulability condition with all tasks in the current solution $\mathcal{S}_{olution}$. We run the test on the starting date values in an increasing order (line 6). If such a starting date exists (line 7), we add the couple (t_j, s_j) to the current solution $\mathcal{S}_{olution}$ (line 8). Even if this starting date does not exist, we continue to check the remaining tasks in $\mathcal{LT}$ in order to find a partial solution and to compute the tasks percentage which can be executed on the same processor.
- Finally, after checking all the tasks in $\mathcal{LT}$, we return the current solution (line 13). If $|\mathcal{S}_{olution}| = n$ then $\mathcal{S}_{olution}$ is a feasible solution. Otherwise, it is a partial solution that contains the subset of tasks which can be scheduled on the same processor.

Note that our algorithm always selects the first starting date value that satisfies the schedulability condition with current solution tasks. Changing the order of checking the tasks starting date values provides a new heuristic. For example, we can modify the order in line 6 from an increasing order to a decreasing one.

Algorithm 4 *Simple* Heuristic

Input: sorted list $\mathcal{LT} = (t_1, \ldots, t_n)$
Output: **feasible** *or* **partial** solution
$\mathcal{S}_{olution} = \{(t_1, r_1)\}$
remove t_1 from $\mathcal{LT}$
for all $t_j \in \mathcal{LT}$ **do**
  for $s_j = r_j$ to $r_j + D_j - C_j$ **do**
    if s_j satisfies equation 6.2 for each $(t_i, s_i) \in \mathcal{S}_{olution}$ **then**
      $\mathcal{S}_{olution} = \mathcal{S}_{olution} \cup \{(t_j, s_j)\}$
      break
    end if
  end for
end for
return $\mathcal{S}_{olution}$

Example 6.5.1 Let us consider the set of strictly periodic communicating tasks $\mathcal{N} = \{t_1, t_2, t_3\}$. Let $\mathcal{E} = \{(t_1, t_3), (t_3, t_2), (t_2, t_1)\}$ be the set of communication relationships between tasks. Table 6.2 represents the tasks parameters. Figure 6.15 depicts the RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ that models the communications between the tasks executions. Our purpose is to verify the system schedulability using Algorithm 4.

t_i	r_i	C_i	D_i	T_i
t_1	0	4	14	18
t_2	0	10	50	144
t_3	0	4	14	54

Table 6.2 Strictly periodic tasks parameters

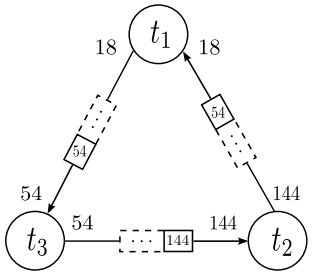


Fig. 6.15 RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$.

Let $\mathcal{LT} = (t_1, t_3, t_2)$ be the tasks sorted list. Tasks are sorted increasingly according to two criteria: the interval length where each task can start its execution $(D_i - C_i)$ and the task period (T_i). We notice that t_1 is prior to t_3 in the sorted list even if they admit equal gaps $(D_1 - C_1 = D_3 - C_3 = 14 - 4 = 10)$. This is due to the fact that the period of t_1 is strictly smaller than the period of t_3 $(T_1 < T_3)$.

In order to build a schedule, we apply Algorithm 4 on the sorted list $\mathcal{LT}$. First, we schedule t_1 with a starting date equals to $s_1 = r_1 = 0$. Then, we verify if there exists a

starting date $s_3 \in [0, 10]$ that allows t_3 to be executed on the same processor executing t_1 (without overlapping). We run the test on the possible values of s_3 in an increasing order. Moreover, we select the first value that verify the schedulability condition with (t_1, s_1). In this example, s_3 is equal to 4 time units. Using the same principle, we verify if there is a starting date $s_2 \in [0, 40]$ that allows t_2 to be scheduled on the processor executing t_1 and t_3. t_2 can be executed on this processor with a starting date equal to 8 time units.

Figure 6.16 depicts the resulting scheduling using Algorithm 4. As we can see, there is no overlapping between the tasks execution (i.e. resource constraints are fulfilled). In addition, we notice that communication constraints between the tasks executions are satisfied, since each task starts its execution during the interval $[r_i, r_i + D_i - C_i]$.

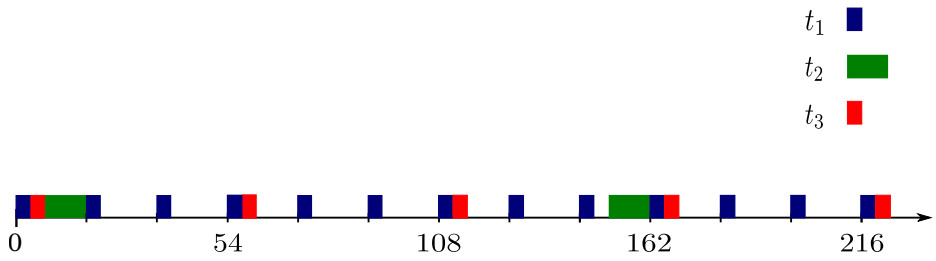


Fig. 6.16 A feasible schedule of the strictly periodic system represented in Table 6.2 and Figure 6.15. The tasks starting dates are respectively equal to $s_1 = 0, s_2 = 8$ and $s_3 = 4$.

Partial solution

Let us modify the previous example by increasing the tasks release dates as follows: $r_1 = 10$, $r_2 = 120$ and $r_3 = 40$ time units. In order to verify the system schedulability on a single processor, we apply Algorithm 4. Let $\mathcal{LT} = (t_1, t_3, t_2)$ be the tasks sorted list. First, we schedule t_1 with a starting date equal to $s_1 = r_1 = 10$ time units. Then, we verify if there is a starting date value of $s_3 \in [40, 50]$ that satisfies the schedulability condition with the starting date of t_1. In this case, s_3 is equal to 40 time units. Finally, we check if there is a starting date $s_2 \in [120, 160]$ that simultaneously satisfies the schedulability condition with t_1 and t_3. We release that such a date does not exist. More precisely, there is no starting date $s_2 \in [120, 160]$ that verifies the schedulability condition with (t_3, s_3). We can deduce that Algorithm 4 provides only a partial solution (see Figure 6.17) composed by the couples $(t_1, 10)$ and $(t_3, 40)$.

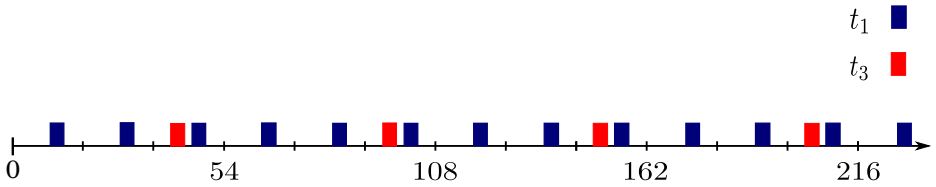


Fig. 6.17 A Partial solution of the scheduling problem using Algorithm 4. The tasks starting dates are respectively equal to $s_1 = 10$ and $s_3 = 40$.

6.5.3 ACAP heuristic

In this part, we present the ACAP (As Close As Possible) heuristic which is also a greedy algorithm. As the the *simple* heuristic, our algorithm takes as input a sorted list of tasks. These latter are sorted increasingly according to their gaps $(D_i - C_i)$ and their periods (T_i). In addition, this algorithm returns a feasible solution (if it is able to find such a solution). Otherwise, it returns a partial solution corresponding to a subset of tasks which can be executed on the same processor. In contrast to the *simple* heuristic approach, a new priority is given to every candidate task in order to be scheduled with the current solution tasks. This priority depends on the candidate task position compared to the other tasks that are already scheduled. In fact, adding this priority seeks to maximize the number of tasks that can executed on a single processor by scheduling the tasks as close as possible.

Scheduling two strictly periodic communicating tasks as close as possible

Let $t_i = (r_i, C_i, D_i, T_i)$ and $t_j = (r_j, C_j, D_j, T_j)$ be two strictly periodic communicating tasks, such that t_i is already scheduled on a processor with a staring date equal to s_i. Let s_j be the unknown starting date of t_j. According to theorem 6.3.1, these tasks can be scheduled on the same processor if and only if equation (6.2) is fulfilled. This equation can be written as follows:

$$s_j - s_i = \delta_{ij} + k_{ij} \cdot g_{ij}, \tag{6.10}$$

where $C_i \leq \delta_{ij} \leq g_{ij} - C_j$ and $k_{ij} = \lfloor \frac{s_j - s_i}{g_{ij}} \rfloor$.

In order to maximize the number of tasks that can be executed on the same processor, we seek to schedule t_j as close as possible to t_i. Hence, we can distinguish two cases:

1. The execution of t_j starts at the end of t_i. This implies that $s_j = s_i + C_i$. In this case, we obtain $\delta_{ij} = (s_j - s_i) \mod g_{ij} = C_i$. Hence, the starting date of t_j can be computed according to the starting date of t_i, as follows:

$$\begin{aligned} s_j - s_i &= C_i + k_{ij} \cdot g_{ij} \\ s_j &= s_i + C_i + k_{ij} \cdot g_{ij}. \end{aligned} \tag{6.11}$$

   In order to fulfil the communication constraints between the tasks executions, s_j must belong to the interval $[r_j, r_j + D_j - C_j]$. Accordingly, we compute the k_{ij} values which verify that t_j starts its execution at the end of t_i, as follows:

$$\begin{aligned} r_j \leq & \quad s_j & \leq r_j + D_j - C_j \\ r_j \leq & \quad s_i + C_i + k_{ij} \cdot g_{ij} & \leq r_j + D_j - C_j \\ r_j - s_i - C_i \leq & \quad k_{ij} \cdot g_{ij} & \leq r_j + D_j - C_j - s_i - C_i \\ \lceil \frac{r_j - s_i - C_i}{g_{ij}} \rceil \leq & \quad k_{ij} & \leq \lfloor \frac{r_j + D_j - C_j - s_i - C_i}{g_{ij}} \rfloor \end{aligned}$$

2. The execution of t_j ends at the beginning of t_i. This implies that $s_i = s_j + C_j$. In this case, we obtain $\delta_{ij} = (s_j - s_i) \mod g_{ij} = g_{ij} - (s_i - s_j) \mod g_{ij} = g_{ij} - C_j$. Hence, the starting date of t_j can be computed according to the starting date of t_i, as follows:

$$\begin{aligned} s_j - s_i &= g_{ij} - C_j + k_{ij} \cdot g_{ij}, \\ s_j &= s_i + g_{ij} - C_j + k_{ij} \cdot g_{ij}, \\ s_j &= s_i - C_j + (k_{ij} + 1) \cdot g_{ij}, \end{aligned} \tag{6.12}$$

   In order to fulfil the communication constraints between the tasks executions, s_j must belong to the interval $[r_j, r_j + D_j - C_j]$. Accordingly, we compute the values of $(k_{ij} + 1)$ which verify that t_j ends its execution at the beginning of t_i, as follows:

$$\begin{aligned} r_j \leq & \quad s_j & \leq r_j + D_j - C_j \\ r_j \leq & \quad s_i - C_j + (k_{ij} + 1) \cdot g_{ij} & \leq r_j + D_j - C_j \\ r_j + C_j - s_i \leq & \quad (k_{ij} + 1) \cdot g_{ij} & \leq r_j + D_j - s_i \\ \lceil \frac{r_j + C_j - s_i}{g_{ij}} \rceil \leq & \quad (k_{ij} + 1) & \leq \lfloor \frac{r_j + D_j - s_i}{g_{ij}} \rfloor \end{aligned}$$

Let $t_1 = (10, 4, 14, 18)$ and $t_2 = (120, 10, 50, 144)$ be two strictly periodic communicating tasks. Let s_1 and s_2 be the tasks starting dates. we suppose that t_1 is already scheduled on a processor with a starting date $s_1 = 10$. Hence, t_2 can start its execution at the end of t_1 if

$$\begin{aligned} s_2 &= s_1 + C_1 + k_{ij} \cdot g_{ij} \\ &= 10 + 4 + k_{ij} \cdot 18, \end{aligned}$$

with $6 \leq k_{ij} \leq 8$. Figure 6.18 illustrates an example where task t_2 begins its execution at the end of t_1 with a starting date $s_2 = 14 + 6 \cdot 18 = 122$ time units. In addition, communication constraints are fulfilled since $s_2 \in [120, 160]$.

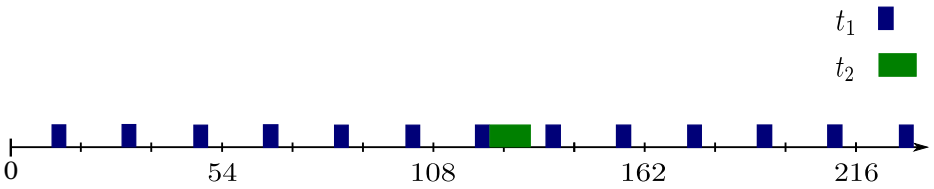


Fig. 6.18 Scheduling task $t_2 = (120, 10, 50, 144)$ at the end of the execution of task $t_1 = (10, 4, 10, 18)$ which is already scheduled with a starting date $s_1 = 10$ time units.

On the other hand, t_2 can end its execution at the beginning of t_1 if

$$\begin{aligned} s_2 &= s_1 - C_2 + (k_{ij} + 1) \cdot g_{ij} \\ &= 10 - 10 + (k_{ij} + 1) \cdot 18, \end{aligned}$$

with $7 \leq (k_{ij} + 1) \leq 8$. Figure 6.19 illustrates an example where task t_2 ends its execution at the beginning of t_1 with a starting date $s_2 = 7 \cdot 18 = 126$ time units. In addition, communication constraints are fulfilled since $s_2 \in [120, 160]$.

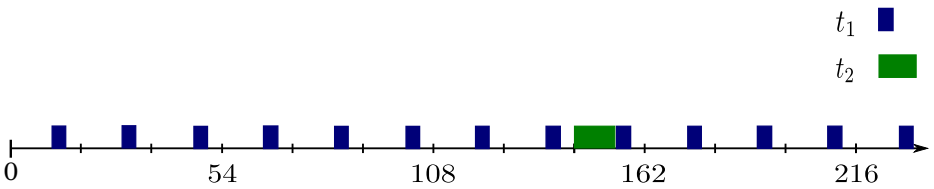


Fig. 6.19 Scheduling task $t_2 = (120, 10, 50, 144)$ such that the end of its execution corresponds to the beginning of task $t_1 = (10, 4, 10, 18)$ which is already scheduled with a starting date $s_1 = 10$ time units.

Our target is to maximize the number of tasks that can be scheduled on the same processor. In order to achieve this goal, we seek to schedule each candidate task as close as possible to the tasks that are already scheduled. Algorithm 5 describes in details this approach as follows:

- As in the *simple* heuristic, we initialize an empty list $\mathcal{S}_{olution}$. We insert in this latter the first task of $\mathcal{LT}$ while setting its starting date to its release date $s_1 = r_1$ (line 3). In addition, this task is removed from $\mathcal{LT}$ (line 4).

- Then, we initialize an empty list $\mathcal{S}_{list}$ (line 6). For each candidate task t_j in $\mathcal{LT}$, we verify if there exists a starting date $s_j \in [r_j, r_j + D_j - C_j]$ which fulfils the schedulability condition with all tasks in the current solution $\mathcal{S}_{olution}$. We run the test on the starting date values in an increasing order (line 8). If a given task t_j does not admit a starting date that satisfies the schedulability condition with all the current solution tasks, then t_j is removed from the list (line 11). In contrast with the *simple* heuristic, we group in $\mathcal{S}_{list}$ all the couples (t_j, s_j) which verify the schedulability condition with current solution tasks (line 14). Moreover, we group in $\mathcal{S}'_{list}$ the couples from $\mathcal{S}_{list}$ which verify equation (6.11) or (6.12) with at least one couple $(t_i, s_i) \in \mathcal{S}_{olution}$ (line 20). In other words, we consider that a couple $(t_j, s_j) \in \mathcal{S}'_{list}$ has a higher priority, since its execution starts at the end of a scheduled task. Moreover, this couple can also have a higher priority if its execution ends at the beginning of a scheduled task. Note that $(t_j, s_j) \in \mathcal{S}'_{list}$ can verify both cases at the same time. This means that a candidate task can begin at the end of a scheduled task and ends at the beginning of another one.

  If several couples in $\mathcal{S}'_{list}$ have the same priority, we choose the couple with the smallest gap (line 22). In addition, if $\mathcal{S}'_{list}$ is empty, we choose from $\mathcal{S}_{list}$ the couple whose gap is minimal (line 24). We add the chosen couple (t_j, s_j) to the current solution $\mathcal{S}_{olution}$ and we remove t_j from $\mathcal{LT}$ (lines 26 and 27). We repeat the whole procedure as long as the list $\mathcal{LT}$ is not empty.

- Finally, once the tasks list $\mathcal{LT}$ is empty, we return the current solution (line 30). If $|\mathcal{S}_{olution}| = n$ then $\mathcal{S}_{olution}$ is a feasible solution. Otherwise, it is a partial solution that corresponds to a tasks subset and their starting dates, that can be scheduled on the same processor.

Algorithm 5 ACAP Heuristic

Input: sorted list $\mathcal{LT} = (t_1, \ldots, t_n)$
Output: **feasible** *or* **partial** solution
$\mathcal{S}_{olution} = \{(t_1, r_1)\}$
remove t_1 from $\mathcal{LT}$
while $\mathcal{LT} \neq \emptyset$ **do**
  $\mathcal{S}_{list} = \emptyset$
  for all $t_j \in \mathcal{LT}$ **do**
    for $s_j = r_j$ to $r_j + D_j - C_j$ **do**
      if $\exists (t_k, s_k) \in \mathcal{S}_{olution}$ such that s_k and s_j **do not** verify equation (6.2) **then**
        if $s_j = r_j + D_j - C_j$ **then**
          remove t_j from $\mathcal{LT}$
        end if
      else
        $\mathcal{S}_{list} = \mathcal{S}_{list} \cup \{(t_j, s_j)\}$
        break
      end if
    end for
  end for
  if $\mathcal{S}_{list} \neq \emptyset$ **then**
    $\mathcal{S}'_{list} = \{(t_j, s_j) \in \mathcal{S}_{list} \mid \exists (t_k, s_k) \in \mathcal{S}_{olution}$, such that s_j and s_k verify equation (6.11) or (6.12)$\}$
    if $\mathcal{S}'_{list} \neq \emptyset$ **then**
      choose the couple $(t_j, s_j) \in \mathcal{S}'_{list}$ whose gap $(D_j - C_j)$ is minimal
    else
      choose the couple $(t_j, s_j) \in \mathcal{S}_{list}$ whose gap $(D_j - C_j)$ is minimal
    end if
    $\mathcal{S}_{olution} = \mathcal{S}_{olution} \cup \{(t_j, s_j)\}$
    remove t_j from $\mathcal{LT}$
  end if
end while
return $\mathcal{S}_{olution}$

Similarly to the *simple* heuristic algorithm, Algorithm 5 always selects the first starting date value that satisfies the schedulability condition with current solution tasks. Changing the order of checking the tasks starting date values provides a new

heuristic. This can be done by modifying the order in line 8 from an increasing order to a decreasing one.

Example 6.5.2 Let us reconsider the system composed of three strictly periodic communicating tasks $t_1 = (10, 4, 14, 18)$, $t_2 = (120, 10, 50, 144)$ and $t_3 = (40, 4, 14, 54)$. Recall that applying Algorithm 4 on this system provides only a partial solution. In order to verify the system schedulability on a single processor, we apply Algorithm 5. Let $\mathcal{LT} = (t_1, t_3, t_2)$ be the tasks sorted list. We schedule t_1 with a starting date equal to its release date $s_1 = r_1 = 10$ time units and we remove it from $\mathcal{LT}$. Then, we compute for each task in $\mathcal{LT}$ its first starting date that fulfils the schedulability condition with $(t_1, 10)$. In this example, the tasks starting dates are respectively equal to $s_3 = 40$ and $s_2 = 122$ time units. We add the couples $(t_3, 40)$ and $(t_2, 122)$ to $\mathcal{S}_{list}$. Moreover, we check for each couple in $\mathcal{S}_{list}$ if its starting date satisfies equation (6.11) or (6.12) with $s_1 = 10$ time units. We notice that $s_3 = 40$ does not satisfy any equation. However, $s_2 = 122$ verifies equation (6.11) with $s_1 = 10$. This means that t_2 starts its execution at the end of t_1. Accordingly, t_2 is given a higher priority and it is scheduled with a starting date equal to $s_2 = 122$ time units. Therefore, we remove t_2 from $\mathcal{LT}$. Finally, We check if there is a starting date $s_3 \in [40, 50]$ that simultaneously verifies the schedulability condition with $(t_1, 10)$ and $(t_2, 122)$. We realize that t_3 can be executed on the same processor with t_1 and t_2 with a starting date $s_3 = 42$ time units.

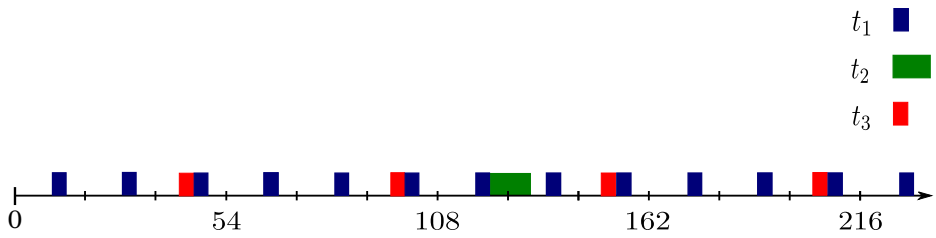


Fig. 6.20 A feasible schedule resulting from applying Algorithm 5 on the perioidic system composed of $t_1 = (10, 4, 14, 18)$, $t_2 = (120, 10, 50, 144)$ and $t_3 = (40, 4, 14, 54)$. The tasks starting dates are respectively equal to $s_1 = 10$, $s_2 = 122$ and $s_3 = 42$ time units.

Figure 6.20 depicts the resulting schedule using Algorithm 5. As we can see, the tasks are scheduled as close as possible. For instance, the first execution of t_3 ends at the beginning of the third execution of t_1. On the other hand, the first execution of t_2 starts at the end of the seventh execution of t_1. In addition, we notice that the resource constraints are fulfilled, since there is no overlapping between the tasks executions. Moreover, communication constraints are fulfilled since each task t_i starts

its execution during the interval $[r_i, r_i + D_i - C_i]$, such as $s_3 = 42$ time units which belongs to $[40, 40 + 14 - 4]$.

Flexible interval case

In the *flexible* intervals case, the starting date of a task $t_i = (r_i, C_i, D_i, T_i)$ is no longer restricted to the interval $[r_i, r_i + D_i - C_i]$. Therefore, we need to know how we might compute the starting date of the new execution interval (r_i^*) knowing that communication constraints between executions must always be respected. Let $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ be a RTSDFG. For each task $t_i \in \mathcal{T}$, we denote by α_i the length of the release date interval to which the starting date of the new execution interval (r_i^*) belongs. More formally,

$$\forall t_i \in \mathcal{T}, r_i^* \in [r_i, r_i + \alpha_i].$$

Computing α_i of a given task $t_i = (r_i, C_i, D_i, T_i)$ must respect the communication constraints between the executions of t_i and that of its adjacent tasks. According to our communication scheme of Section 5.1 in Chapter 5, the data consumption can only be done at the release date of the receiving task. As the starting date of the new execution interval (r_i^*) is greater or equal to the task release date (r_i), the communications constraints between the task and its predecessors are always fulfilled. Therefore, the α_i computation depends only on the task's release dates and those of its successors. More precisely, it depends on the minimal latency separating the task's executions and those of its successors. According to Theorem 5.2.2 in Section 5.2 of Chapter 5, the minimum latency between two periodic communicating tasks t_i and t_j, such that t_i is the emitting task and t_j the receiving one, can be computed as follows:

$$\mathcal{L}_{min}(t_i, t_j) = r_j - r_i + \lambda - D_i,$$

where $\lambda = \lceil \frac{r_i - r_j + D_i}{gcd_a} \rceil \cdot gcd_a$ and $gcd_a = \gcd(T_i, T_j)$.

Our approach computes the interval length α_i of a task t_i while taking into account that t_i can admit successors that are already scheduled. Therefore, our algorithm takes as input a candidate task $t_i = (r_i, C_i, D_i, T_i)$ and a current solution $\mathcal{S}_{olution} = \{(t_j, r_j^*, s_j),$ such that t_j is already scheduled$\}$. It returns the interval length (α_i) to which r_i^* belongs.

Algorithm 6 Computation of the length of the release date interval: α_i

1: **Input**: task $t_i = (r_i, C_i, D_i, T_i)$, current solution $\mathcal{S}_{olution}$
2: **Output**: α_i
3: $\mathcal{S}_{ucc}(t_i)$: set of successors of t_i
4: **if** $\mathcal{S}_{ucc}(t_i) = \emptyset$ **then**
5: $\quad \alpha_i \leftarrow \mathcal{M} - r_i - (R_i - 1) \cdot T_i - D_i$
6: **else**
7: $\quad$ **for all** $t_j \in \mathcal{S}_{ucc}(t_i)$ **do**
8: $\quad\quad$ **if** $t_j \in \mathcal{S}_{olution}$ **then**
9: $\quad\quad\quad \mu_j = r_j^*$
10: $\quad\quad$ **else**
11: $\quad\quad\quad \mu_j = r_j$
12: $\quad\quad$ **end if**
13: $\quad\quad \mathcal{L}_{min}(t_i, t_j) = \mu_j - r_i + \lambda - D_i \quad$ with $\lambda = \lceil \frac{r_i - \mu_j + D_i}{gcd_a} \rceil \cdot gcd_a$
14: $\quad$ **end for**
15: $\quad \mu_i \leftarrow \min\{\mathcal{L}_{min}(t_i, t_j)\}\ \forall t_j \in \mathcal{S}_{ucc}(t_i)$
16: $\quad \alpha_i \leftarrow \min\{\mu_i,\ \mathcal{M} - r_i - (R_i - 1) \cdot T_i - D_i\}$
17: **end if**
18: **return** α_i

Algorithm 6 describes in details our approach, as follows:

- First, we check in line 4 if the successors list is empty. This means that there is no restriction on the interval length value and communication constraints are always fulfilled. We denote by $\mathcal{M}$ the date before which each task $t_i \in \mathcal{T}$ is executed at least R_i (task repetition factor) times. In our case, we compute an upper bound on this date in the following way:

$$\mathcal{M} = \max_{\forall t_i \in \mathcal{T}} \{r_i\} + \beta \cdot R_i \cdot T_i,$$

with $\beta \in [2, n]$ where n is the number of tasks ($|\mathcal{T}|$). In addition, we assume that the R_ith interval of length D_i ends at the instant $\mathcal{M}$. Therefore, the length of the release date interval (α_i) can be computed as follows:

$$r_i + \alpha_i + (R_i - 1) \cdot T_i + D_i = \mathcal{M}$$
$$\alpha_i = \mathcal{M} - r_i - (R_i - 1) \cdot T_i - D_i$$

- In contrast, if the successors list is not empty, lines 7 to 12 classify the successors tasks whether they are already scheduled or not. Accordingly, line 13 represents the minimum latency computation between the task and its successor. In other words, we compute the minimum latency $\mathcal{L}_{min}(t_i, t_j)$ according the starting date of the new execution interval (r_i^*) if the successor task is already scheduled. Otherwise, we compute this latency according to the successor release date. In line 15, we compute μ_i which is equal to the minimum $\mathcal{L}_{min}(t_i, t_j)$ between t_i and its successors. In line 16, we calculate the length of the release date interval (α_i) by computing the minimum between μ_i and $\mathcal{M} - r_i - (R_i - 1) \cdot T_i - D_i$.

After computing the interval length α_i, we compute the starting date of the new execution interval r_i^*. Our approach consists of finding the task starting date s_i (if it exists), and according to its value, we compute the interval starting date. Our algorithm takes as input a candidate task $t_i = (r_i, C_i, D_i, T_i)$, the length of its interval release date α_i and a current solution $\mathcal{S}_{olution}$. It returns the starting date of the new execution interval (r_i^*).

Algorithm 7 Computation of the starting date of the new execution interval: r_i^*

1: **Input**: task $t_i = (r_i, C_i, D_i, T_i)$, current solution $\mathcal{S}_{olution}$, interval length α_i
2: **Output**: r_i^*
3: $\mathcal{P}_{rec}(t_i)$: set of predecessors of t_i
4: **if** $\exists s_i \in [r_i, r_i + D_i - C_i + \alpha_i]$ such that t_i can be scheduled on the same processor with the current solution tasks **then**
5: **if** $\exists t_j \in \mathcal{P}_{rec}(t_i)$ such that $t_j \notin \mathcal{S}_{olution}$ **then**
6: $r_i^* = \min(r_i + \alpha_i, s_i)$
7: **else if** $s_i + C_i > r_i + D_i$ **then**
8: $r_i^* = s_i + C_i - D_i$
9: **else**
10: $r_i^* = r_i$
11: **end if**
12: **end if**
13: **return** r_i^*

Algorithm 7 computes the starting date of the new execution interval. First, line 3 checks if there is a task starting date (s_i) which belongs to the interval $[r_i, r_i + D_i - C_i + \alpha_i]$, such that t_i can be scheduled on the same processor with the current solution tasks. We distinguish two cases:

1. Task t_i admits a predecessor ($t_j \in \mathcal{P}_{rec}(t_i)$) which is not scheduled yet. If such a predecessor does exist (line 5), we seek to increase the starting date value of the new execution interval (r_i^*). This increases the possibility of finding a starting date (s_j) that allows the predecessor task (which is not scheduled yet) to be executed on the same processor with the current solution tasks. Therefore, r_i^* can be computed (line 6) by finding the minimum between $r_i + \alpha_i$ and the task starting date s_i.

2. Task t_i does not admit predecessors or admits predecessors that are already scheduled. In this case, we check whether the task is partially or totally executed outside the interval $[r_i, r_i + D_i]$. If this occurs, the starting date of the new execution interval is equal to $s_i + C_i - D_i$ (line 8). Otherwise, the task is totally executed within the interval $[r_i, r_i + D_i]$ and r_i^* is equal to r_i (line 10).

Our purpose is to solve the mono-processor scheduling problem for the *flexible* intervals case using the *simple* or ACAP heuristics. In order to achieve our target, we modify Algorithm 4 or 5 as follows:

- We verify whether the first task of the sorted list admits predecessors. If this occurs, we compute the length of the release date interval (α_i) using Algorithm 6. Moreover, we schedule this task with a starting date s_i equal to $r_i + \alpha_i$. Otherwise, we schedule this task with a starting date equals to its release date r_i. In both cases, the starting date of the new execution interval r_i^* is equal to the task starting date.

- After scheduling the first task, we seek to schedule the remaining tasks in $\mathcal{LT}$. Therefore, at the beginning of each iteration of Algorithm 4, we compute for the candidate task the length of its release date interval (α_i) (using Algorithm 6). However, at the beginning of each iteration of Algorithm 5, we compute for every remaining task in $\mathcal{LT}$ the length of its release date interval (α_i) (using Algorithm 6). In both cases, we need to find the first date which allows the task to be scheduled on the same processor with the current solution tasks. In order to do that, we run the test on the starting date value in the interval $[r_i, r_i + D_i - C_i + \alpha_i]$.

- Once the task's starting date s_i is fixed, we compute its starting date of the new execution interval r_i^* according to s_i using Algorithm 7.

Example 6.5.3 Let us consider the set of strictly periodic communicating tasks $\mathcal{N} = \{t_1, t_2, t_3, t_4\}$. Let $\mathcal{E} = \{(t_1,t_2),(t_1,t_2),(t_1,t_3),(t_2,t_3),(t_3,t_4),(t_3,t_5)\}$ be the set of communication relationships between tasks. Table 6.2 shows the tasks parameters. Figure 6.15 depicts the RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ that models the communications between the tasks executions. Our purpose is to schedule this system using the ACAP heuristic for the *flexible* intervals case.

t_i	r_i	C_i	D_i	T_i
t_1	160	20	50	240
t_2	10	10	20	30
t_3	80	10	30	120
t_4	20	20	50	60
t_5	60	10	40	120

Table 6.3 Strictly periodic tasks parameters

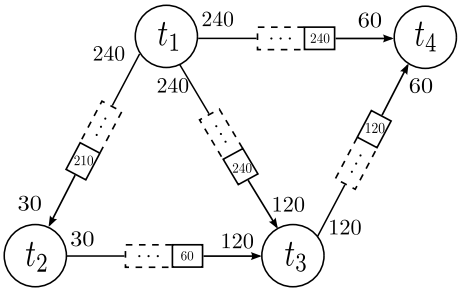


Fig. 6.21 RTSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$.

Let $\mathcal{P}_{rec}(t_i)$ and $\mathcal{S}_{ucc}(t_i)$ be respectively the set of predecessors and successors of task t_i. Let $\mathcal{LT} = (t_2, t_3, t_4, t_1)$ be the tasks sorted list. We notice that t_2 (the first task in $\mathcal{LT}$) admits t_1 as predecessor. Consequently, we apply Algorithm 6 in order to compute α_2 knowing that $\mathcal{S}_{ucc}(t_2) = t_3$. As t_3 is not scheduled yet, we compute $\mathcal{L}_{min}(t_2,t_3)$ according to the release date of t_3 as follows:

$$\begin{aligned}\mathcal{L}_{min}(t_2,t_3) &= r_3 - r_2 + \lambda - D_2 \\ &= 80 - 10 + \lceil \frac{10 - 80 + 20}{30} \rceil \cdot 30 - 20 \\ &= 20 \text{ time units.}\end{aligned}$$

On the other hand, we consider $\mathcal{M}$ a date before which each task is executed at least R_i (its repetition factor) times. An upper bound on this date can be calculated in the following way:

$$\begin{aligned}\mathcal{M} &= \max_{\forall t_i \in \mathcal{T}} \{r_i\} + \beta \cdot R_i \cdot T_i \\ &= 160 + \beta \cdot 240,\end{aligned}$$

with $\beta \in [2,4]$. In this example, we consider $\beta = 4$. Hence, the length of the release date interval is equal to

$$\begin{aligned}\alpha_2 &= \min\{\mathcal{L}_{min}(t_2,t_3), \mathcal{M} - r_2 - (R_2-1)\cdot T_2 - D_2\}\\ &= \min\{20, 1120 - 10 - (8-1)\cdot 30 - 30\}\\ &= \min\{20, 870\} = 20 \text{ time units.}\end{aligned}$$

Once α_2 computed, we schedule t_2 with a starting date of the new execution interval equals to

$$s_2 = r_2^* = r_2 + \alpha_2 = 10 + 20 = 30 \text{ time units.}$$

We add $(t_2, 30, 30)$ to the current solution $\mathcal{S}_{olution}$ and we remove t_2 from $\mathcal{LT}$. At the beginning of the first iteration, we apply Algorithm 6 on each task in $\mathcal{LT} = (t_3, t_4, t_1)$. The computed values of α_i are $\alpha_3 = 30$, $\alpha_4 = 870$ and $\alpha_1 = 0$ time units. Hence, s_3, s_4 and s_1 are tested in the intervals $[80, 130]$, $[20, 920]$ and $[160, 190]$, respectively. We compute for each task its first starting date that fulfils the schedulability condition with $(t_2, 30, 30)$. In this iteration, these dates are respectively equal to $s_3 = 80$, $s_4 = 40$ and $s_1 = 160$ time units. Moreover, we verify for every task if its starting date satisfies equation (6.11) or (6.12) with $s_2 = 30$ time units. We notice that $s_3 = 80$ verifies equation (6.12). However, $s_4 = 40$ and $s_1 = 160$ fulfil both equations at the same time. This means that t_4 and t_1 can be exactly executed between two successive executions of t_2. More specifically, their starting dates (s_4 and s_1) coincide with the end date of an execution of t_2 and their end dates ($s_4 + C_4$ and $s_1 + C_1$) coincide with the starting date of the next execution of t_2. In addition, t_4 has a smaller gap than t_1 ($D_4 - C_4 < D_1 - C_1$). Accordingly, t_4 is given a higher priority and it is scheduled with $s_4 = 40$ time units. After computing s_4, we apply Algorithm 7 in order to calculate the starting date of the new execution interval r_4^*. As t_4 admits t_1 and t_3 as predecessors (which are not scheduled yet), r_4^* can be computed as follows:

$$\begin{aligned}r_4^* &= \min(r_4 + \alpha_4, s_4)\\ &= \min(20 + 870, 40)\\ &= 40 \text{ time units.}\end{aligned}$$

This procedure is repeated as long as the list $\mathcal{LT}$ is not empty. At the second iteration, we schedule t_1 with $s_1 = 190$ and $r_1^* = 160$ time units (since $\mathcal{P}_{rec}(t_1) = \emptyset$ and the task is totally executed in the interval $[160, 210]$). Finally, we schedule t_3 with $s_3 = 130$ time

units. As t_3 is executed outside the interval $[80,110]$ and all its predecessor are already scheduled, r_3^* is equal to $s_3+C_3-D_3=130+10-30=110$ time units.

Figure 6.22 illustrates the feasible schedule resulting from applying Algorithm 5 for the *flexible* case. The tasks' starting dates are $s_1=190$, $s_2=30$, $s_3=130$ and $s_4=40$ time units. The starting dates of the new execution intervals are $r_1^*=160$, $r_2^*=30$, $r_3^*=110$ and $r_4^*=40$ time units. We notice that the resource constraints are fulfilled since there is no overlapping between the tasks executions. Moreover, communication constraints are fulfilled since each task t_i starts its execution during the interval $[r_i^*, r_i^*+D_i-C_i]$, such as $s_4=40$ time units which belongs to $[40,40+90-20]$.

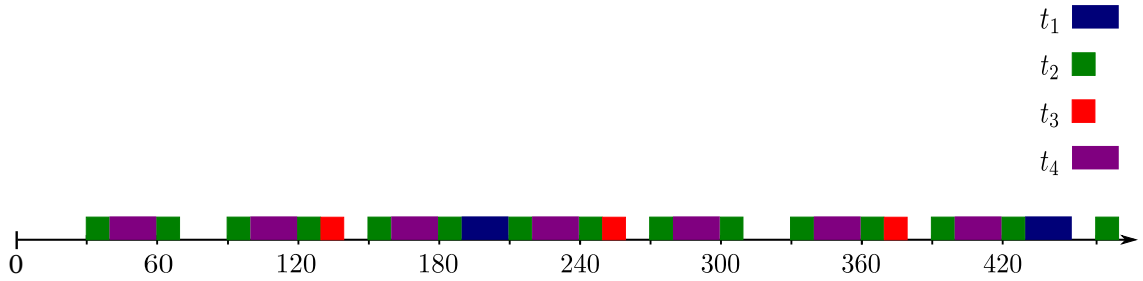


Fig. 6.22 A feasible schedule of the strictly periodic system represented in Table 6.3 and Figure 6.21. The tasks starting dates are $s_1=190$, $s_2=30$, $s_3=130$ and $s_4=40$ time units. In addition, the starting dates of the new execution intervals are $r_1^*=160$, $r_2^*=30$, $r_3^*=110$ and $r_4^*=40$ time units.

6.6 Conclusion

This chapter has introduced an overview of strictly periodic tasks and their characteristics. According to the communication model defined in Chapter 5, we proved that scheduling two periodic communicating tasks is equivalent to schedule two independent tasks. These latter can only be executed between their release dates and their deadlines. In order to solve the mono-processor scheduling problem of a strictly periodic communicating system, we propose an optimal (exact) algorithm using MILP formulations. Two cases were treated: the *fixed* and *flexible* interval cases. In the *fixed* case, tasks execution is only restricted to the interval beginning at its release date and ending at its deadline. However, in the *flexible* case, tasks can admit new execution intervals whose lengths are equal to their relative deadlines. The scheduling problem is know to be NP-complete in strong sense [79]. This means, obtaining feasible schedule is practicable only for small instances. In order to solve this issue, we proposed three heuristics: linear programming relaxation, *simple* and ACAP heuristics. For

the second and the third heuristic if no feasible solution is found, a partial solution is computed. This solution corresponds to a subset of tasks that can be executed on the same processor.

The next chapter presents the experimental results of this thesis. It is devoted to evaluating the methods that compute the worst case system latency using the RTSDFG formalism. In addition, it presents several experiments dedicated to evaluate the performance of our methods that solve the mono-processor scheduling problem.

Chapter 7

Experimental Results

This chapter presents the experimental results of this thesis. Section 7.1 present briefly the random generator used to generate our RTSDFGs instances. Section 7.2 is devoted to evaluate our methods that compute the worst case latency of a RTSDFG. Section 7.3 evaluates the performance of our algorithms that solve the mono-processor scheduling problem of strictly periodic communicating systems. Section 7.4 concludes the chapter.

7.1 Graph generation: Turbine

Bodin et al.[19] recently developed a data flow random generator "Turbine". It generates randomly a Phased Computation Graph (PCG in short) [116] which is an extension of the SDFG model. Turbine requires several parameters to be fixed in order to generate instances. Some of these parameters are directly related to the graph structure, such as the total number of tasks $|\mathcal{T}|$ and their degrees. The other parameters, as the average value of the repetition vector $\left(R_{\mathcal{T}} = \frac{\sum_{t \in \mathcal{T}} R_t}{|\mathcal{T}|}\right)$, control the random generation of production and consumption rates.

Turbine is implemented using Python and the graph library NetworkX that manage the graph data structure. Random generation is composed of three steps. The first step generates a random graph with nodes and arcs. The second step computes the consumption and production rates of the graph. The last step computes a live initial marking using the Linear Programming Solver (GLPK).

In our study, we added to Turbine two parameters: the task release date (r_i) and its relative deadline (D_i). The initial marking of each buffer $a = (t_i, t_j)$ is computed as follows:

$$M_0(a) = T_j + \lambda - gcd_a,$$

where $\lambda = \lceil \frac{r_i - r_j + D_i}{gcd_a} \rceil \cdot gcd_a$ and $gcd_a = \gcd(T_i, T_j)$. The relative deadline was generated randomly such that its value do not exceed the production (or consumption) rate. The duration of each task was generated randomly in such way that it does not exceed the task relative deadline.

7.2 Latency evaluation

This section is devoted to evaluate our methods that compute the worst case latency of a RTSDFG. We expose different results for the exact evaluation, upper and lower bounds. These results are obtained by modifying several parameters such as the average repetition factor and the number of tasks. Finally, we verify the quality of our bounds and the complexity of our algorithms.

We generated randomly acyclic RTSDFG instances. The buffer production and consumption rates were chosen uniformly from the following set

$$\mathcal{F} = \{T_i : \mathcal{R} > 0,\ \mathcal{R} \equiv 0 \mod T_i\},$$

where $\mathcal{R}$ is a fixed strictly positive integer and $\mathcal{F}$ the set of its divisors. In addition, the tasks' release dates were randomly generated from the set $\{0, \ldots, lcm\}$ with $lcm = \operatorname{lcm}_{\forall t_i \in \mathcal{T}}(T_i)$.

Complexity

The transformation from a RTSDFG to an equivalent graph $\mathcal{G}'$, which presents all the precedence and dependence constraints between the tasks executions, uses a mathematical equation of complexity $\mathcal{O}(R_{\mathcal{T}})$ where $R_{\mathcal{T}}$ is the average repetition factor of the tasks. However, transforming a RTSDFG into an weighted graph $\mathcal{G}_{max}$ (or $\mathcal{G}_{min}$), which represents the required time to transfer the data in the worst (or best) case scenario, uses mathematical equation of complexity $\mathcal{O}(1)$. Furthermore, the different evaluation processes of the worst case latency use the same algorithm that computes the length of the longest path in a DAG. Its complexity is $\mathcal{O}(|\mathcal{T}| + |\mathcal{A}|)$. Therefore, complexity of the latency exact evaluation is $\mathcal{O}(R_{\mathcal{T}} \cdot (|\mathcal{T}| + |\mathcal{A}|))$, while the complexity of computing its bounds is $\mathcal{O}(|\mathcal{T}| + |\mathcal{A}|)$.

7.2.1 Computation time

The experiments were performed on a Intel(R) Core2Duo P8600@2.40GHz using 2 GB of RAM. We set up parallel calculations for some instances on a server with 48 cores. Each algorithm was launched on 100 instances. Each one returned the average value of the exact latency evaluation, upper and lower bound as well as their computation times. Four graphs sizes have been chosen: *Small* for 100 tasks, *Medium* for 500 tasks, *Large* for 1000 tasks and *Huge* for 10000 tasks. Input and output degrees of a task belong to the set $\{1, 2, 3, 4, 5\}$.

$R_{\mathcal{T}}$	Computation time (s)		
	Exact evaluation	Upper bound	Lower bound
5	22.23	5.24	4.93
25	107.14	5.20	5.12
45	212.71	5.22	5.05
75	337.72	5.23	4.98
125	557.25	5.32	5.11

Table 7.1 Average computation time of latency evaluation methods for Huge RTSDFGs with respect to their average repetition factors.

In Table 7.1, the size of the RTSDFG is fixed to 10000 tasks. Average repetition factors were varied in order to present the latency computation time for the exact evaluation, upper and lower bounds. By increasing the average repetition factor of the tasks, the time required to compute the exact value of the worst-case latency increases (linearly). Contrariwise, the time needed to compute the bounds remains approximately constant (equal to 5 seconds).

$\lvert\mathcal{T}\rvert$	Computation time (s)		
	Exact evaluation	Upper bound	Lower bound
Small	3.40	0.15	0.15
Medium	16.76	0.46	0.45
Large	34.11	0.93	0.93
Huge	337.72	5.23	4.98

Table 7.2 Average computation time of latency evaluation methods for RTSDFGs according to their size $|\mathcal{T}|$. The average repetition factor is fixed to 75.

Through varying the sizes of RTSDFGs, table 7.2 and 7.3 present a comparison between the average computation times of an exact value and its bounds. The average repetition factors were fixed to 75 and 750 respectively. In both cases, the average time

$\|\mathcal{T}\|$	Computation time (s)		
	Exact evaluation	Upper bound	Lower bound
Small	37.58	0.23	0.23
Medium	196.43	0.62	0.61
Large	481.52	0.97	0.97
Huge	3898.26	5.04	5.62

Table 7.3 Average computation time of latency evaluation methods for RTSDFGs according to their size $|\mathcal{T}|$. The average repetition factor is fixed to 750.

required to compute the latency bounds remains reasonable (in the range of seconds) compared to that of the exact value. For instance, computing the worst case latency exact value of a Huge graph with a average repetition factor is 750, requires on average more than one hour. However, the computing its lower and upper bounds requires approximately 5 seconds.

7.2.2 Quality of bounds

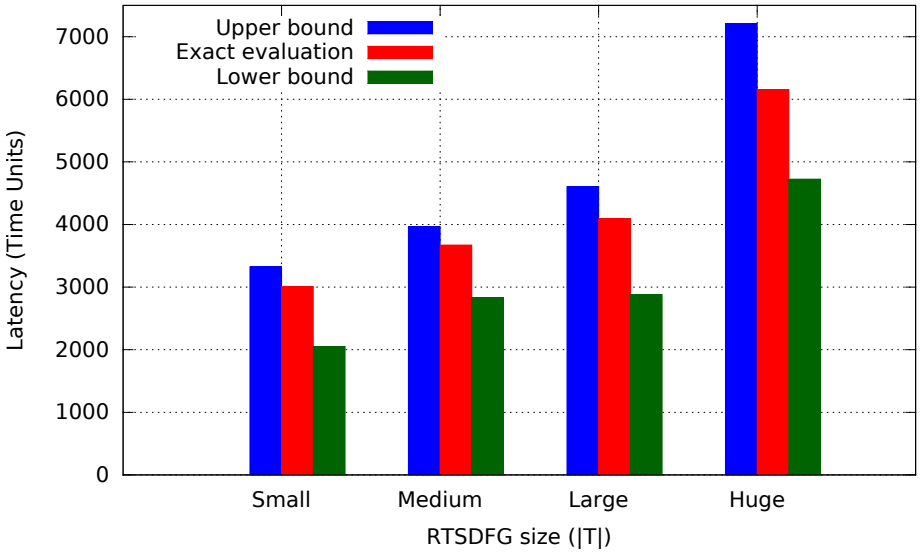
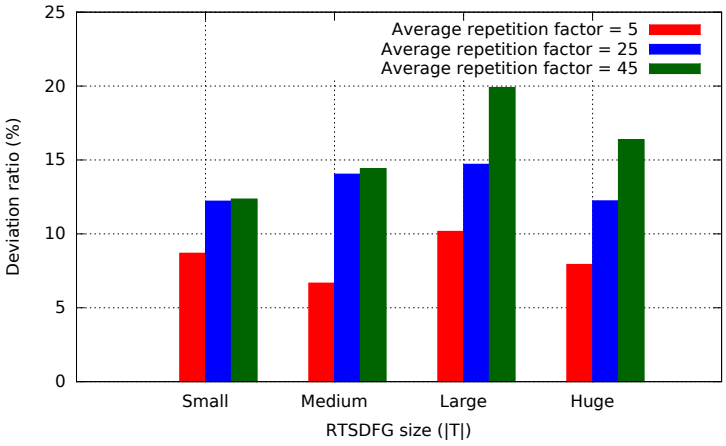


Fig. 7.1 Average latency value using the evaluation methods with respect to the graph size $|\mathcal{T}|$.The average repetition was fixed to 125.

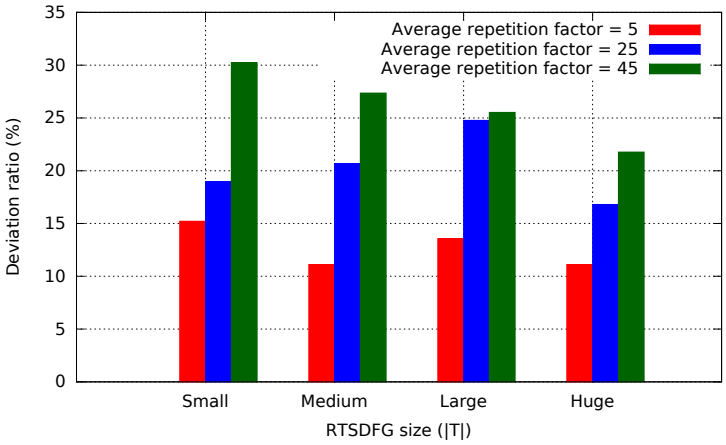
The histogram in Figure 7.1 represents the latency evaluations with respect to the graph size. The average repetition factor was fixed to 125. The analysis of these results shows that the gap between the exact evaluation and its upper bound on one

hand is smaller than the gap between the same exact value and its lower bound on the other hand. This is due to the fact that the exact value and the upper bound are both computed in the worst case scenario, whereas, the lower bound is computed in the best case scenario.

In this experiment, we evaluated the impact of the average repetition factor ($R_{\mathcal{T}}$) on evaluating the worst case system latency. Therefore, we generated acyclic RTSDFG without release dates ($\forall t_i \in \mathcal{T}, r_i = 0$).



(a) Upper bound.



(b) Lower bound.

Fig. 7.2 Deviation ratio between the latency exact value and its bounds. Deviation ratio of upper bound is equal to $dr_{up} = \frac{\text{upper}-\text{exact}}{\text{exact}} \times 100\%$. Deviation ratio of lower bound is equal to $dr_{lw} = \frac{\text{exact}-\text{lower}}{\text{exact}} \times 100\%$.

Deviation ratio is the percentage of gaps between the latency exact value and its bounds. We denote by dr_{up} as the deviation ratio of the upper bound which is

computed as follows:

$$dr_{up} = \frac{\text{upper} - \text{exact}}{\text{exact}} \times 100\%.$$

However, we denote by dr_{lw} as the deviation ratio of the lower bound which is computed as follows:

$$dr_{lw} = \frac{\text{exact} - \text{lower}}{\text{exact}} \times 100\%.$$

Through varying the average repetition factor for different size of RTSDFGs, the histograms in Figure 7.2a and 7.2b depict the deviation ratios of the latency upper and lower bounds respectively. For different size of RTSDFGs, we notice that deviation ratios increase with the average repetition factors. This is due to the time required to transfer the data between two communicating tasks in the worst (or best) case scenario. The computation of this duration is strongly dependant from the production (or consumption) rates whose generation is tightly related to the average repetition factor. In addition, we can deduce that the deviation ratio between the exact value and the upper bound ranges between 10 and 15 %. However, the deviation ratio between the exact value and the lower bound varies between 20 and 30 %.

7.3 Mono-processor scheduling

An important challenge is solving scheduling problems of complex real-time systems. These problems can lead to large-scale mathematical formulations which are intractable, even with modern optimization solvers. In other words, there are many scheduling problems that cannot easily be solved, even with commercial mixed-integer programming software packages such as Gurobi Optimizer.

This section presents several experiments dedicated to evaluate the performance of our methods that solve the mono-processor scheduling problem of strictly periodic communicating systems. First, we describe the generation of the task parameters. Then, we expose several results concerning the resolution of the scheduling problem with an exact algorithm for different types of instances. Finally, we present the results obtained by applying our heuristics on the same instances.

7.3.1 Generation of tasks parameters

Using the modified version of Turbine, we generated cyclic and acyclic instances of RTSDFGs. Parameters of each task have been generated as follows:

- Task utilization factors (u_i, $1 \leq i \leq n$) have been generated based using UUnifast algorithm [18]. The complexity of this algorithm is $\mathcal{O}(n)$ and the generated utilization factors are characterized by a uniform distribution.

- Harmonic tasks periods, i.e. each period is multiple of all other periods with lesser value, were uniformly chosen form the set $\{T_i : T_{i+1} = k \cdot T_i\}$ with the period ratio $k \in \{2, 3, 4, 5\}$. The value of the first period (T_1) was equal to 500. However, non-harmonic periods were chosen uniformly from the set $\{2^x \cdot 3^y \cdot 50 \ : x, y \in [0, 4]^2\}$, which is inspired from [38]. In both cases, the size of the set containing the periods values was fixed to 5.

- Task worst case execution time was computed according to the task period and its utilisation factor, as follows: $C_i = \lceil T_i \cdot u_i \rceil$.

- Task relative deadline was generated randomly such that its value is greater or equal to the task execution time on one hand, and its lesser or equal to the task period on the other hand ($C_i \leq D_i \leq T_i$).

- Task release date was generated randomly from the set $\{0, \ldots, lcm\}$ where $lcm = \operatorname{lcm}_{i \in \{1,\ldots,n\}}(T_i)$.

t_i	1	2	3	4	5	6	7	8	9	10
T_i	7200	2700	2700	600	7200	150	150	900	150	2700
C_i	690	198	216	36	210	6	12	54	12	6
r_i	4367	20989	19680	6240	5696	12841	5347	13844	10472	9059
D_i	3876	1154	2254	412	5847	33	32	588	115	1345

Table 7.4 Example of 10 non-harmonic tasks. Each task is characterized by a period T_i, worst case execution time C_i, release date r_i and relative deadline D_i. The lcm between the task periods is equal to 21600.

Table 7.4 presents an example of a non-harmonic instance temporal parameters. The graph size was equal to 10 tasks and the system utilization was 0.6.

7.3.2 Optimal algorithm

In this subsection, we present some experimental results to evaluate the performance of our exact algorithms for both *fixed* and *flexible* cases. The MILP formulations for the mono-processor scheduling problem are solved using the mixed-integer programming solver Gurobi (Academic license - for non-commercial use only).

The size of the graph (number of tasks) was chosen from the set $\{5, 10, 15, 20, 25, 30\}$. The system utilization was varied from 0.1 to 1. Input and output degrees of a task belong to the set $\{1, 2, 3, 4, 5\}$. We fixed a solver time limit of 600 seconds. Otherwise, the solver does not stop seeking whether a feasible solution exists or not, even after four hours. The instances for which the solver does not return a response during 600 seconds were called "solver failure instances".

Each algorithm was launched on 1000 instances. Each one returned the average value of the acceptance ratio, the percentage of infeasible instances, the percentage of solver failure instances as well as their computation times. The experiments were performed on an Intel(R) Core(TM) i3-3130M CPU @ 2.60GHz using 4 GB of RAM. We set up parallel calculations for some instances on a server with 40 cores.

Harmonic periods

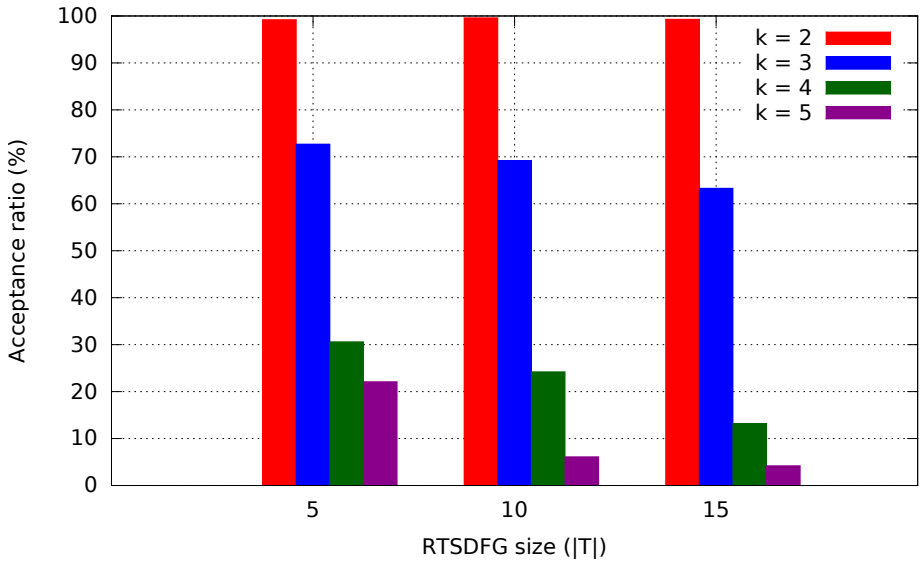


Fig. 7.3 MILP acceptance ratio on harmonic tasks generated with different periods ratios. The system utilization was fixed to 0.1. The RTSDFG size ($|\mathcal{T}|$) was equal to 5, 10 and 15 tasks.

In order to evaluate the impact of the periods ratio on the MILP acceptance ratio, we generated harmonic instances by increasing the periods ratio $k \in \{2, 3, 4, 5\}$. The system utilization was fixed to 0.1 and the RTSDFG size was equal to 5, 10 and 15 tasks. Figure 7.3 depicts the MILP acceptance ratio on these harmonic instances. We can find that the incrementation of the periods ratio leads to the decrease of the

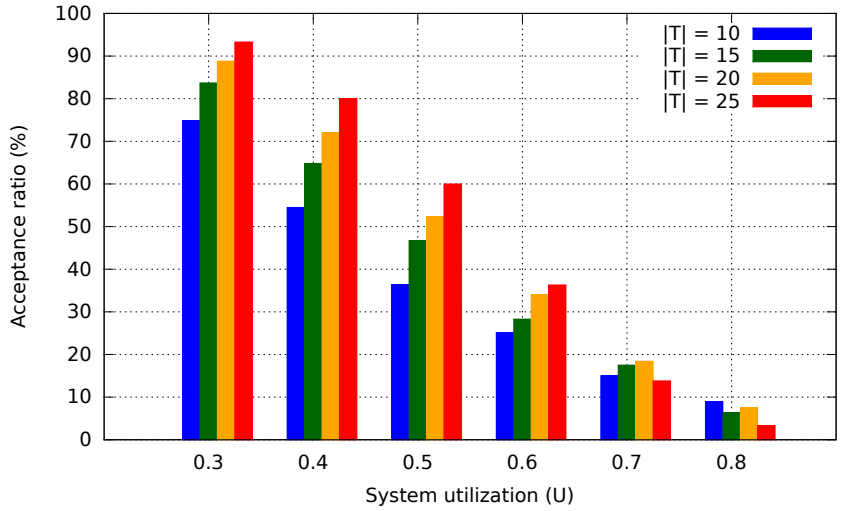
MILP acceptance ratio. For example, when the RTSDFG size is equal to 10 tasks, the acceptance ratio decreases from 99.6% for $k = 2$ to 6.1% for $k = 5$. In fact, the increase of the period ratio induces an increase in the period values as well as an increase in the gap between the period values. In addition, this increase in the periods values leads to larger worst execution times ($C_i = \lceil T_i \cdot u_i \rceil$). For these reasons, finding a feasible schedule is more and more difficult. Therefore, the MILP acceptance ratio decreases by increasing the periods ratio.

In the sequel, harmonic instances were generated with a period ratio equals to 2.

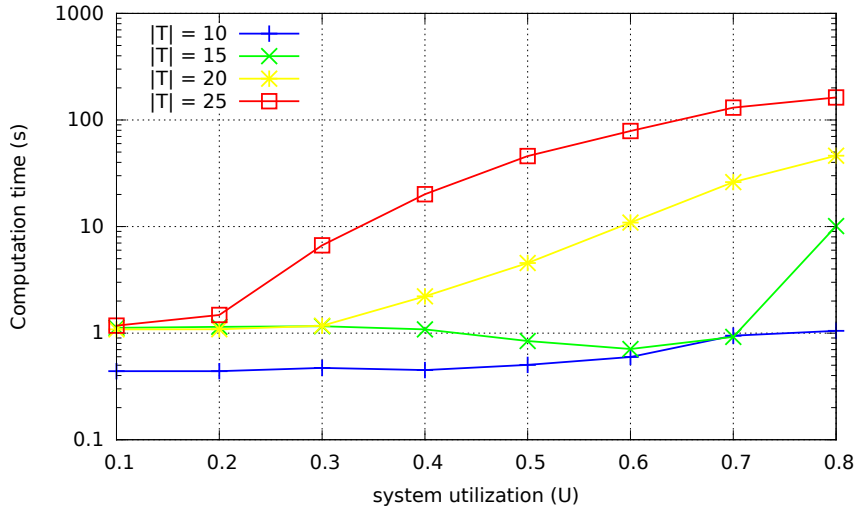
The histogram in Figure 7.4a represents the MILP acceptance ratio with respect to the RTSDFG size (number of tasks) and the system utilization. For a system utilization between 0.3 and 0.6, we notice that the MILP acceptance ratio increases with increasing size of RTSDFG. In fact, the increase in the number of tasks leads to a reduction in the tasks worst case execution times. This allows to find more feasible schedules. However, this is no longer the case once the system utilization is equal to 0.8. For example, the acceptance ratio decreases from 15.3% for $|\mathcal{T}| = 10$ to 8.9% for $|\mathcal{T}| = 15$.This is due to the tasks worst case execution times for $|\mathcal{T}| = 15$ which are not small enough to be scheduled on the same processor.

For a fixed size of RTSDFG, we notice that the acceptance ratio decreases with the increase of the system utilization. For instance, when the RTSDFG size is equal to 20 tasks, the acceptance ratio decreases from 88.8% for $U = 0.3$ to 17.5% for $U = 0.8$. Indeed, the tasks worst execution times grow with the increase of the system utilization.

Figure 7.4b demonstrates that the average computation time to find a feasible solution increases with the incrementation of the RTSDFG size and the system utilization. For example, when the system utilization is 0.7, the time required to find a feasible solution increases from 0.92 seconds for $|\mathcal{T}| = 10$ to 130.68 seconds for $|\mathcal{T}| = 25$. This is due to the number of the MILP variables that grows with the increase of the RTSDFG size. Moreover, when the RTSDFG size is equal to 20 tasks, the time required to find a feasible solution increases from 1.08 seconds for $U = 0.4$ to 40.46 seconds for $U = 0.8$. This means that the increase in the worst execution time (due to the system utilization increment) requires the solver to use a larger amount of time in order to find a feasible solution.



(a) MILP acceptance ratio (%).



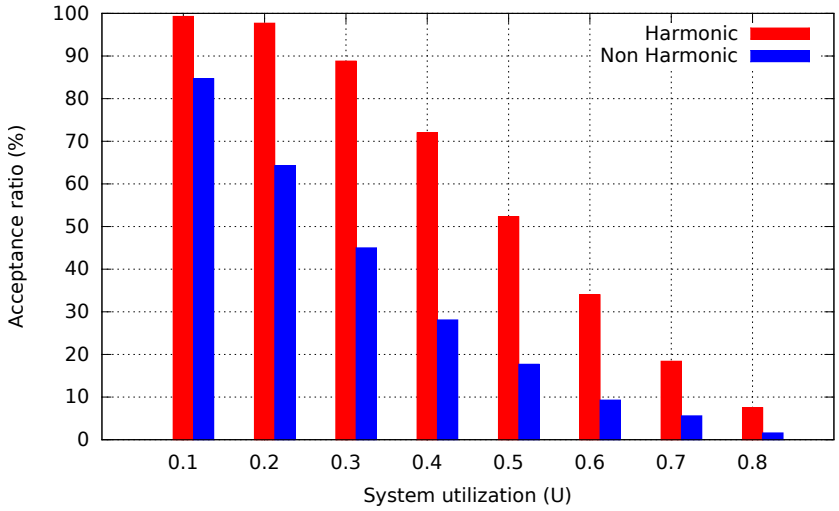
(b) Average computation time to find a feasible schedule.

Fig. 7.4 Experimental results on harmonic instances of RTSDFGs with different RTSDFG sizes ($|\mathcal{T}|$) and system utilizations (U).

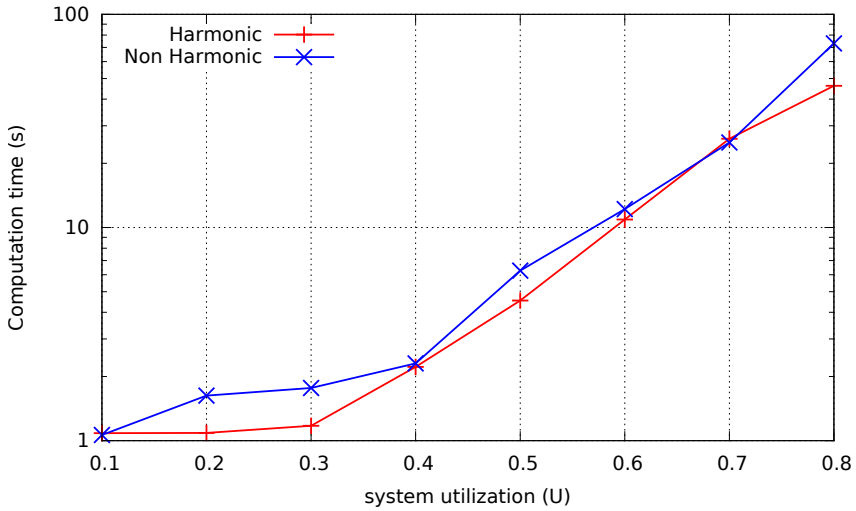
Harmonic versus Non Harmonic periods

In order to evaluate the effects of the tasks periods on our exact method, we generated harmonic and non-harmonic instances with different system utilizations. The RTSDFG size was fixed to 20 tasks. Figure 7.5a presents a comparison between the MILP acceptance ratio for harmonic and non-harmonic instances with respect to the system utilization. We notice that the acceptance ratio on harmonic instances is greater than that on non-harmonic ones. For example, when the system utilization is 0.3, the

acceptance ratio for harmonic instances is equal to 88.8% while that for non-harmonic ones is equal to 45.0%. This is mainly due to the resource constraints that depend on the greatest common divisor (gcd) value of each pair of periods. Indeed, the gcd of a pair of harmonic periods is equal to the minimum period. However, the gcd of a pair of non-harmonic periods is less or equal to the minimum period.



(a) MILP acceptance ratio (%).



(b) Average computation time to find a feasible solution.

Fig. 7.5 Experimental results on harmonic and non-harmonic instances with different system utilizations. The RTSDFG size $|\mathcal{T}|$ was fixed to 20 tasks.

Figure 7.5b shows the average computation time required to find a feasible solution for harmonic and non-harmonic instances. We can see that the average time required

by the solver to find a feasible schedule is slightly higher in the non-harmonic case than harmonic one. This means that the solver requires more time to allocate the tasks to a narrow (restricted) space.

Acyclic versus Cyclic RTSDFGs

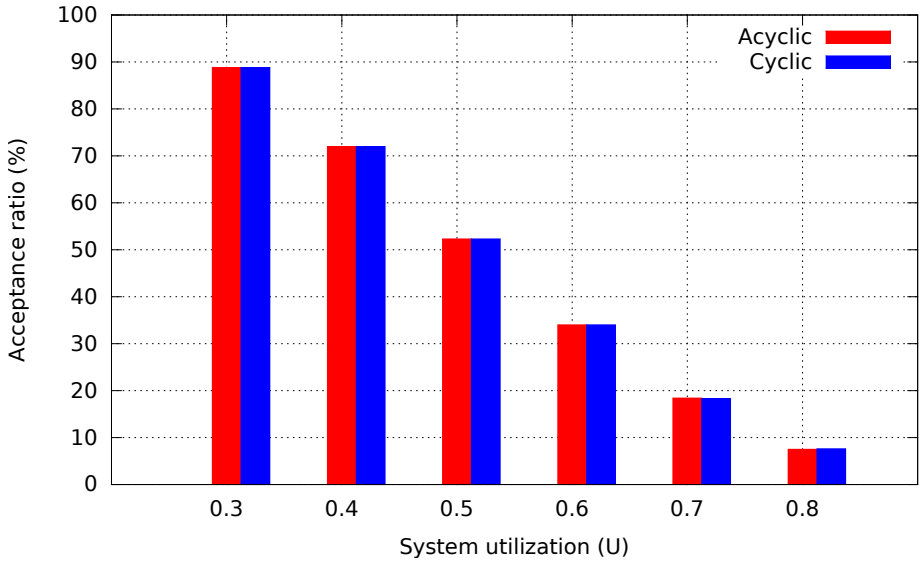


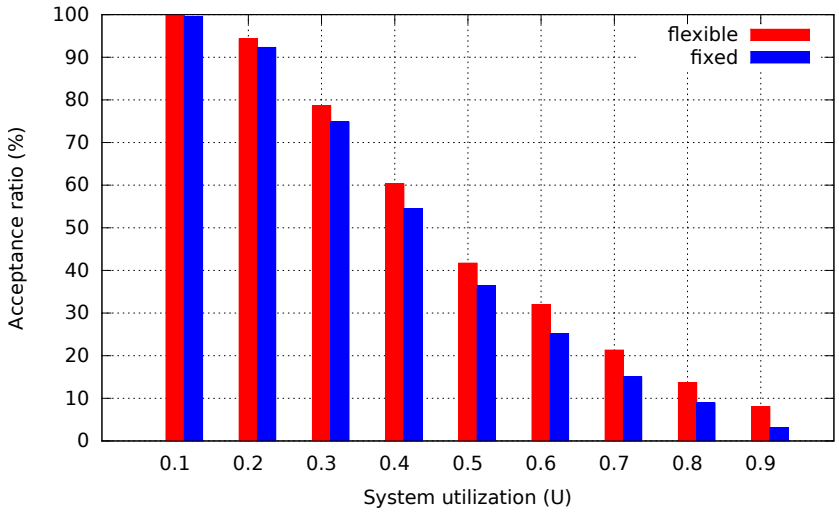
Fig. 7.6 MILP acceptance ratio on harmonic acyclic and cyclic instances with respect to the system utilization. The graph size $|\mathcal{T}|$ was fixed to 20 tasks.

In order to evaluate the impact of the graph topology on the MILP acceptance ratio, we started by generating acyclic RTSDFGs. Then, we generated cyclic RTSDFGs by adding arcs to the acyclic instances (the others parameters remain unchanged such as the tasks periods and their execution times). The RTSDFG size was fixed to 20 tasks. Figure 7.6 represents the MILP acceptance ratio on acyclic and cyclic instances. We notice that the acceptance ratio on acyclic instances is equal to that on cyclic ones. Indeed, communication constraints between the tasks executions are fulfilled when each task is executed between its release date and its deadline. Hence, we can deduce that our exact method is independent from the graph topology.

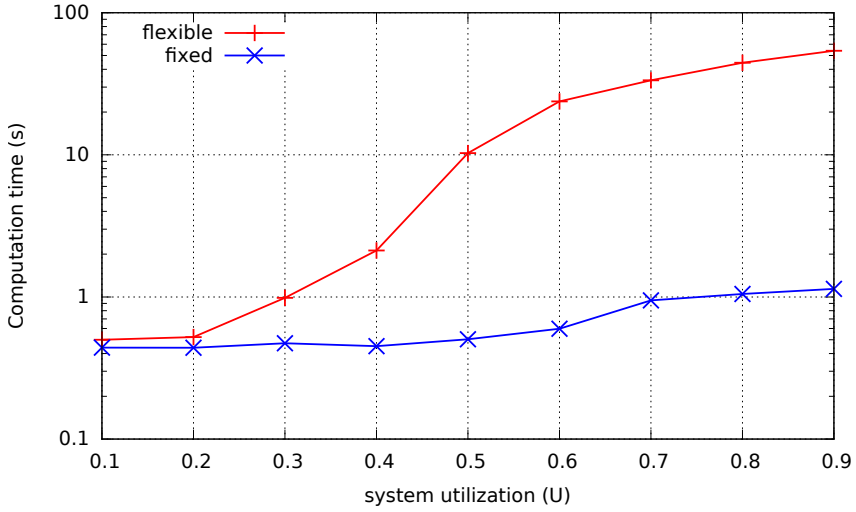
Fixed versus *Flexible* intervals cases

In this experiment, we generated harmonic instances with different system utilizations. The RTSDFG size was fixed to 10 tasks. We applied the exact algorithms on the same

instances for both *fixed* and *flexible* cases. Figure 7.7a presents the MILP acceptance ratio for both cases. We notice that the acceptance ratio for the *flexible* case is higher than the *fixed* one. The gap between these ratios varies between 2% and 7%. In fact, scheduling a task in the *flexible* case is no longer restricted to the interval between its release date and its deadline as in the *fixed* case. This allows the solver to find more feasible solutions in the *flexible* case than the the *fixed* one.



(a) MILP acceptance ratio for *fixed* and *flexible* cases with respect to the system utilization.



(b) Average computation time to find a feasible solution in *fixed* and *flexible* cases.

Fig. 7.7 Experimental results on harmonic instances for *fixed* and *flexible* cases. The RTSDFG size $|\mathcal{T}|$ was fixed to 10 tasks.

Figure 7.7b shows the average computation time to find a feasible schedule in both *flexible* and *fixed* cases. We notice that the time required by the solver to find a feasible solution in the *flexible* case is higher than in the *fixed* one. For example, when the system utilization is 0.8, the solver requires on average 1.04 seconds to find a feasible schedule in the *fixed* case while it requires 44.48 seconds in the *flexible* case. This is due to the addition of the interval starting date (r_i^*) variable in the *flexible* case.

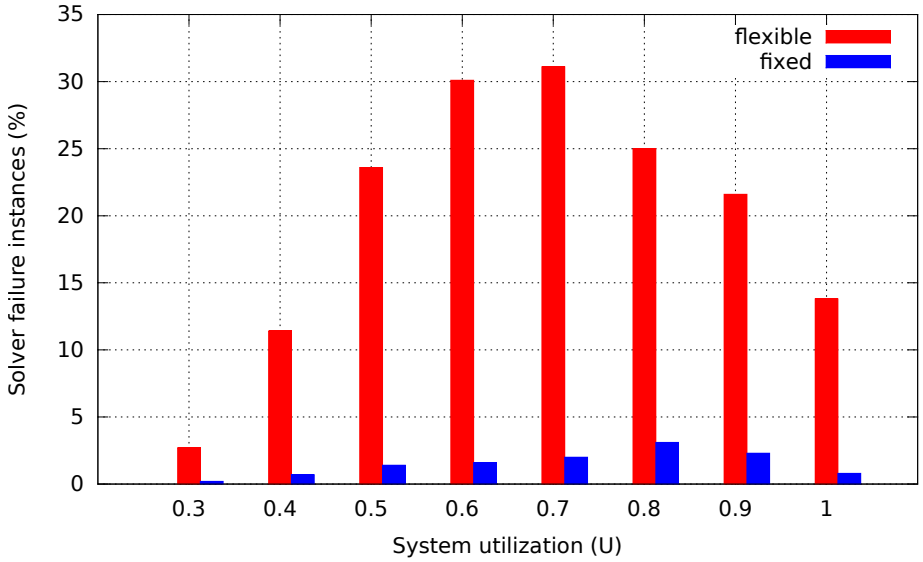


Fig. 7.8 Percentage of solver failure instances for *fixed* and *flexible* cases with respect to the system utilization. The RTSDFG size $|\mathcal{T}|$ was fixed to 20 tasks.

As we mentioned before, the solver time limit was fixed 600 seconds. Figure 7.8 presents a comparison between the percentage of solver failure instances for the *flexible* and *fixed* cases. The RTSDFG size was fixed to 20 tasks. We can see that the percentage of solver failure instances in the *flexible* case is considerably higher than in the *fixed* case. This is due to the combinatorial explosion generated by the addition of variables in the *flexible* case. For a system utilization under 0.7 for the *flexible* case and 0.8 for the *fixed* case, we notice that the percentage of solver failure instances increases with the increase of the system utilization. For instance, this percentage increases from 2.7% for $U = 0.3$ to 31.1% for $U = 0.7$ in the *flexible* case. Moreover, the percentage of solver failure instances decreases with a system utilization beyond 0.7 for the *flexible* case and 0.8 for the *fixed* case. In fact, instances with a system utilization beyond these values admit less and less feasible schedules. Therefore, we can deduce that in

our interest to verify whether a given instance admits a feasible schedule for the *fixed* case before applying the MILP for the *flexible* case.

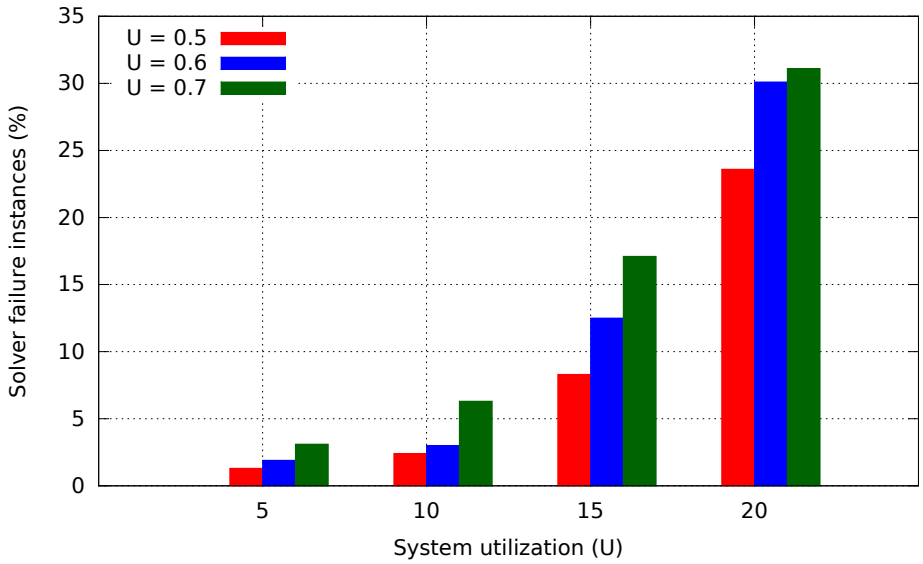


Fig. 7.9 Percentage of solver failure instances for *flexible* case with respect to the graph size. The system utilizations were equal to $0.5, 0.6$ and 0.7.

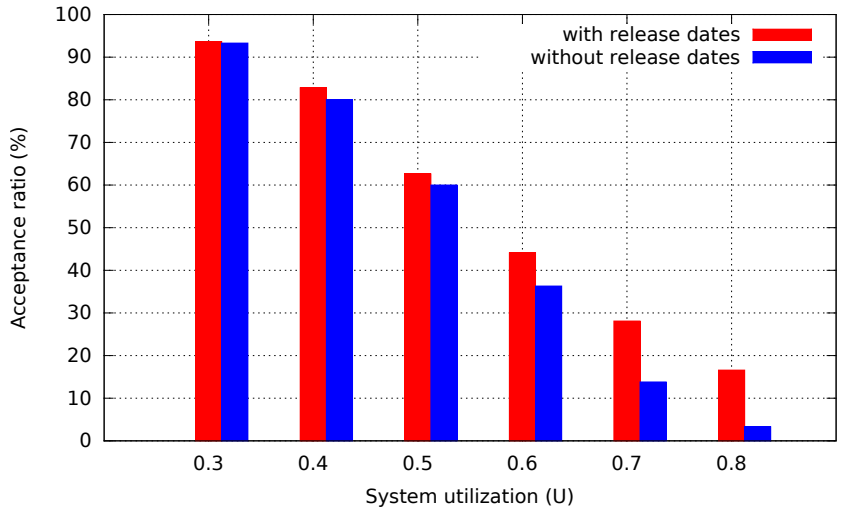
On the other hand, Figure 7.9 shows the percentage of solver failure instances in the *flexible* case with respect to the graph size. The system utilizations were fixed to 0.5, 0.6 and 0.7. We notice that the percentage of solver failure instances increases with the increase of the RTSDFG size. For example, when the system utilization is 0.5, the percentage of solver failure instances increases from 1.3% for $|\mathcal{T}| = 5$ to 23.6% for $|\mathcal{T}| = 20$. This is mainly due to the number of variables that increases by increasing the RTSDFG size.

Release dates

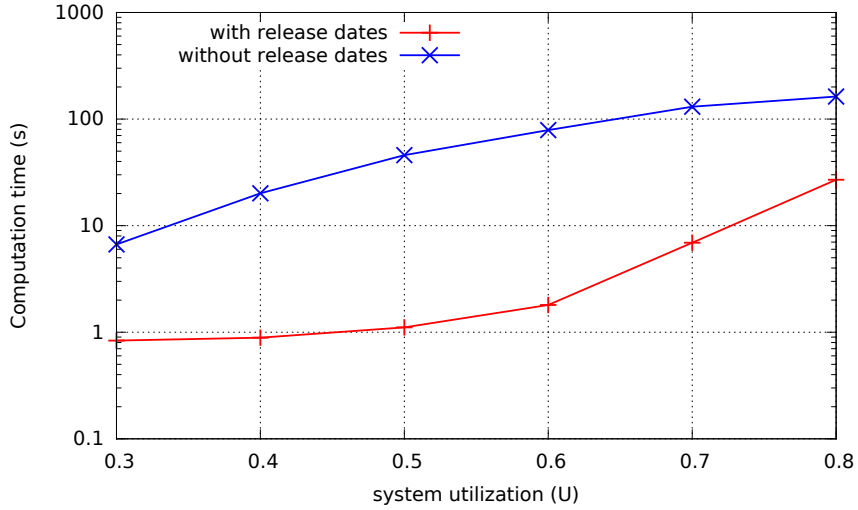
We now evaluate the impact of the release date parameter on our method. To this end, we started by generating harmonic instances without release dates ($\forall t_i \in \mathcal{T}, r_i = 0$). Then, we generated instances with release dates by adding to each RTSDFG task a random release date as follows:

$$\forall t_i \in \mathcal{T}, r_i \in \{0, \ldots, lcm\},$$

where $lcm = \underset{i \in \{1,\ldots,n\}}{\text{lcm}}(T_i)$. The RTSDFG size was fixed to 25 tasks and other parameters remain unchanged.



(a) MILP Acceptance ratio (%).



(b) Average computation time to find a feasible solution.

Fig. 7.10 Experimental results on harmonic instances with and without release dates. The RTSDFG size $|\mathcal{T}|$ was fixed to 25 tasks.

Figure 7.10a represents a comparison between the MILP acceptance ratios on instances with and without release dates. As we can see, the acceptance ratio on instances with release dates is higher than the acceptance ratio on instances without release dates. This is mainly due to the time limitation (solver time limit = 600 seconds). We also notice that the gap between these acceptance ratios increases with

the increase of the system utilization. This is due to the increase of the tasks worst case execution times.

Figure 7.10b depicts the average computation time to find a feasible schedule for instances with and without release dates. We notice that the solver requires more time to find a feasible solution for instances without release dates than instances with release dates. For example, when the system utilization is fixed to 0.7, the solver requires on average 6.9 seconds to find a feasible schedule for instances with release dates while it requires 130.68 seconds for instances without release dates. This is because the solver requires more time to allocate the tasks to a restricted space.

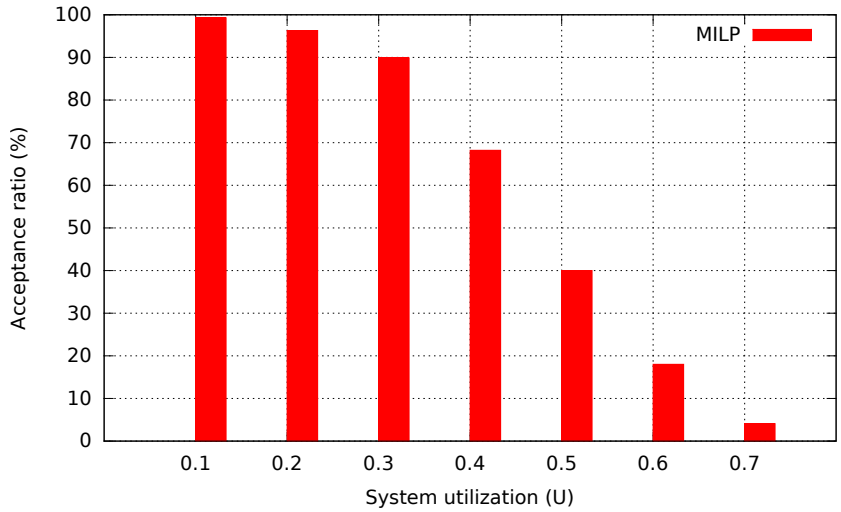
7.3.3 Heuristics

In this subsection, we expose the experimental results of our heuristics: LP relaxation, *simple* and ACAP. Some instances that cannot be solved by one heuristic, can be scheduled by the other heuristics. Consequently, we propose a mega-heuristic "Mega" that combines several heuristics. Given an RTSDFG instance, this mega-heuristic applies sequentially the *simple* and ACAP heuristics for the *fixed* case. In addition, it applies the LP relaxation, *simple* and ACAP heuristics for the *flexible* case. In both cases, it either returns a feasible solution or a partial solution that contains a set of tasks that can be scheduled on the same processor. In order to evaluate their performances compared to the exact methods, heuristics were only tested on feasible instances (that admit feasible schedules using the exact methods) and solver failure instances. Finally, heuristics were applied on all the instances without any exception in order to compare their performances for the *fixed* and *flexible* cases.

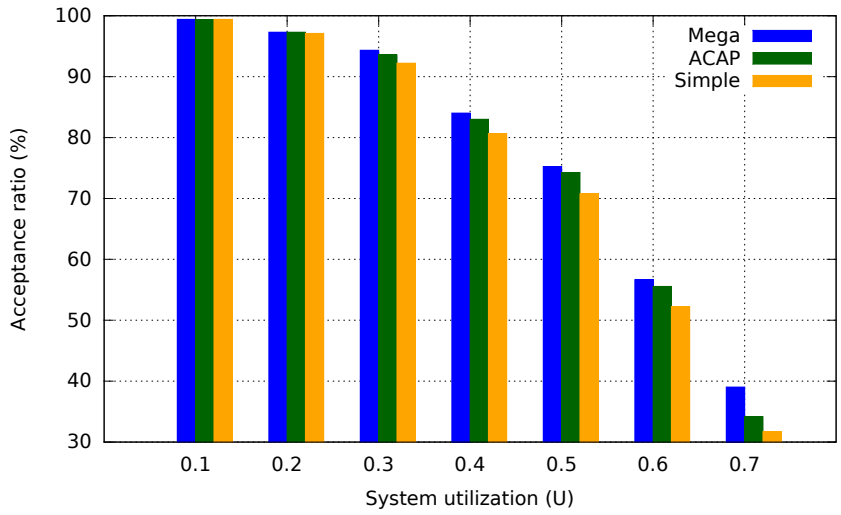
Complexity

In the *simple* heuristic, the main computation part of the algorithm is composed of double loops. The inner loop iterates on the interval of length $(D_i - C_i)$ in order to find a starting date that allows the task to be scheduled on the processor. The outer loop iterates on all the tasks in the system ($|\mathcal{T}|$). Hence, the *simple* heuristic complexity is $\mathcal{O}(|\mathcal{T}| \cdot l_{max})$ where $l_{max} = \max(D_i - C_i)$. However, the ACAP heuristic complexity is $\mathcal{O}(|\mathcal{T}|^2 \cdot l_{max})$, since a third loop was added to compute the list of tasks that verify the schedulabily condition with tasks that are already scheduled.

***Fixed* case**



(a) MILP acceptance ratio for the *fixed* case.



(b) Heuristics acceptance ratio on feasible instances.

Fig. 7.11 Experimental results on harmonic instances without release dates. The RTSDFG size $|\mathcal{T}|$ was fixed to 30 tasks.

In this experiment, we generated harmonic instances without release dates. The system utilization was varied between 0.1 and 0.7. The RTSDFG size was fixed to 30 tasks. Using the exact method for the *fixed* case, instances were sorted into feasible, infeasible and solver failure. Figure 7.11a presents the MILP acceptance ratio for the *fixed* case. Figure 7.11b gives the heuristics acceptance ratios on feasible instances. We notice that the mega-heuristic have a higher acceptance ratio compared to those of the *simple*

and ACAP heuristics. For the three heuristics, we notice that the acceptance ratios decrease with the increase of the system utilisation. For example, the Mega acceptance ratio decreases from 99.4% for $U = 0.1$ to 39.02% for $U = 0.7$. This is mainly due to the increase of the tasks worst execution times.

Figure 7.12 depicts the average computation time to find a feasible schedule using different methods. We notice that our methods have a similar changing tendency, i.e. their average computation times increase with the increase of the system utilization. This is due to the amount of computation that grows with the incrementation of the tasks worst execution times. However, our heuristics have computation times considerably lower than the exact method (MILP). For example, when the system utilization is 0.5, the average computation times of the exact method, Mega, ACAP and *simple* heuristics are respectively equal to 154.31, 3.02, 15.43 and 2.37 seconds. Therefore, we deduce that the Mega heuristic is 51.09 times faster than the exact method (MILP) with an acceptance ratio of 75.25%. On the other hand, we notice the ACAP heuristic is more time consumption than the *simple* heuristic. Hence, the experimental results are compatible with the computational complexity.

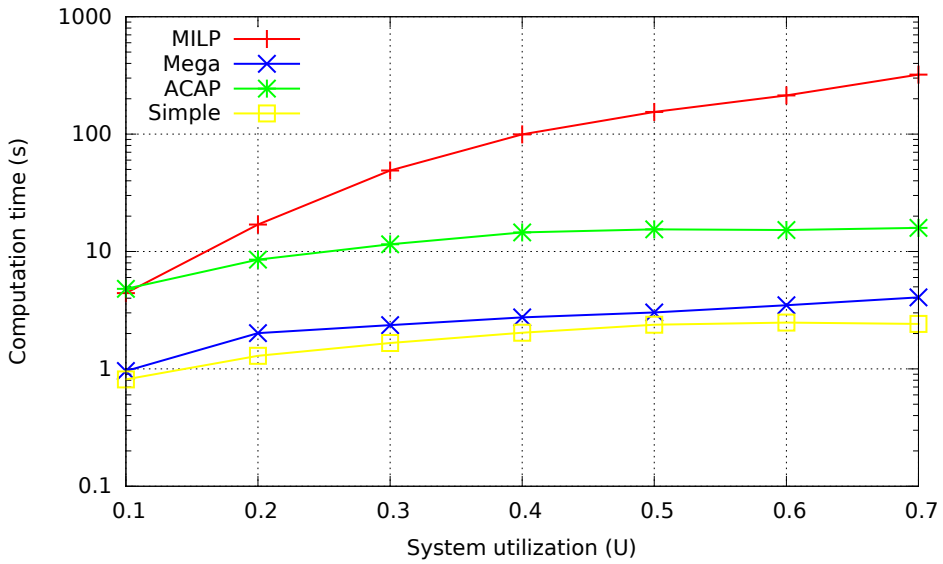
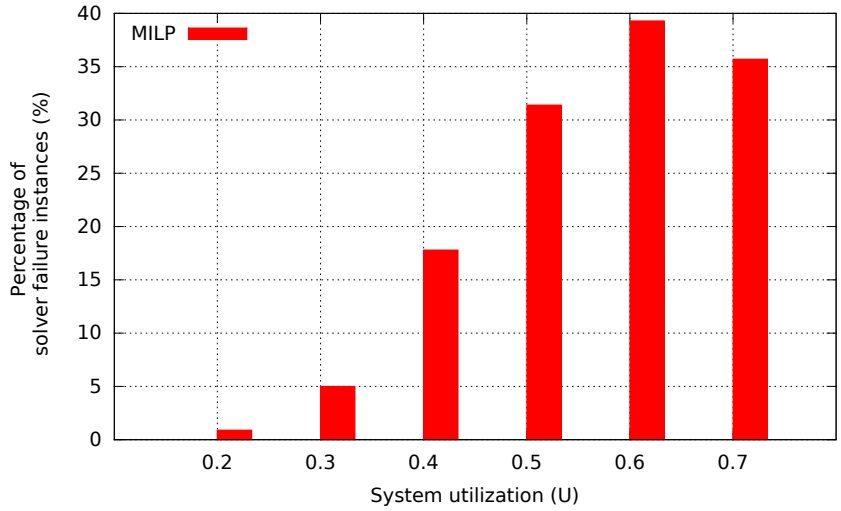


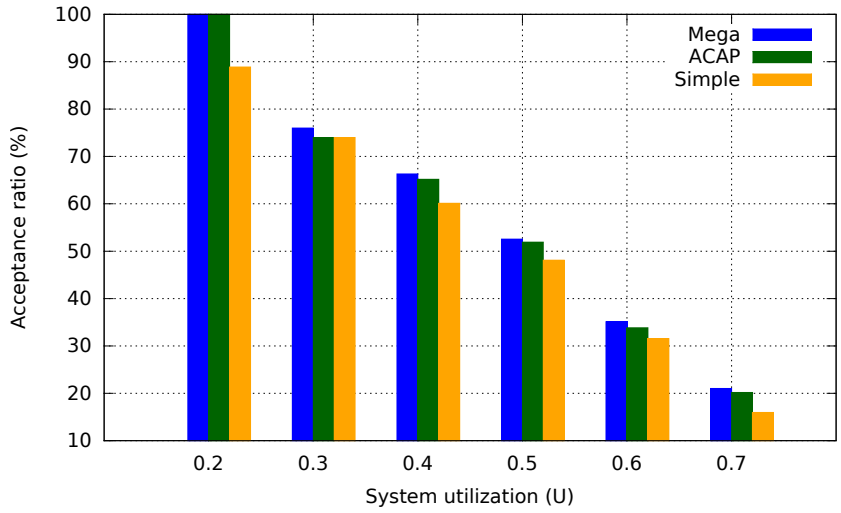
Fig. 7.12 Average computation time to find a feasible solution. MILP represents our exact method while Mega, ACAP and *simple* represent our heuristics.

In the experiment of Figure 7.13b, we apply our heuristics on the solver failure instances. Figure 7.13a shows the percentage of these instances with respect to the system utilization. Similarly to the previous case, the heuristics acceptance ratios

decrease with the increase of the system utilization. For example, the Mega acceptance ratio decreases from 100% for $U = 0.2$ to 21.01% for $U = 0.7$. We can deduce that our heuristics are able to find feasible solutions for these instances with a computation time in the range of seconds. However, the MILP solver is not able to provide any solution with a computation time below 600 seconds.



(a) Percentage of solver failure instances.



(b) Heuristics acceptance ratio on solver failure instances.

Fig. 7.13 Results obtained by applying our heuristics on the solver failure instances. The RTSDFG size $|\mathcal{T}|$ was fixed to 30 tasks.

We recall that our heuristics return a partial solution if they are not able to compute a feasible one. This partial solution contains the set of tasks that can be scheduled

on the same processor. Figure 7.14 gives the percentage of schedulable tasks with respect to the system utilization. We notice that this percentage decreases with the incrementation of the system utilization. This is because the tasks worst case execution times grow with the system utilization. In addition, we deduce that the percentage of schedulable tasks ranges between 96% and 86% using the Mega heuristic.

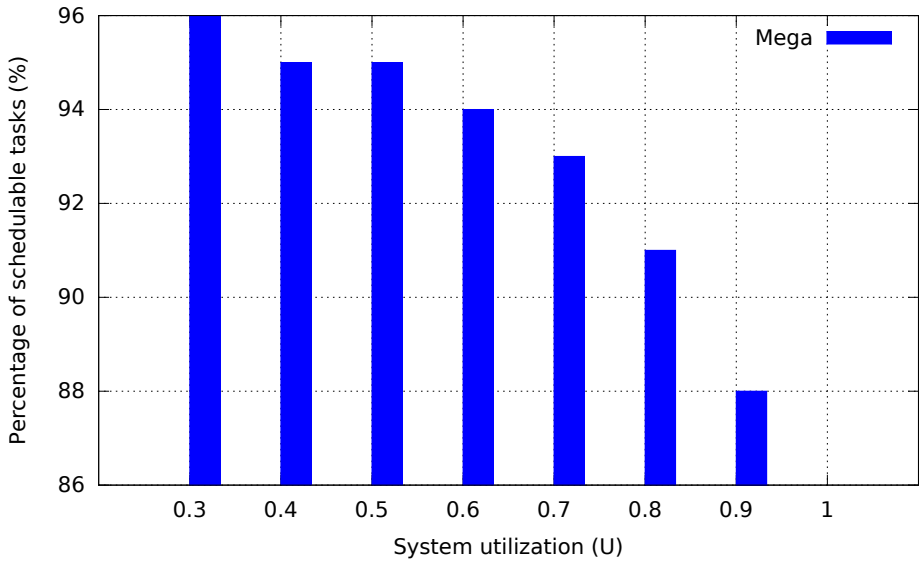
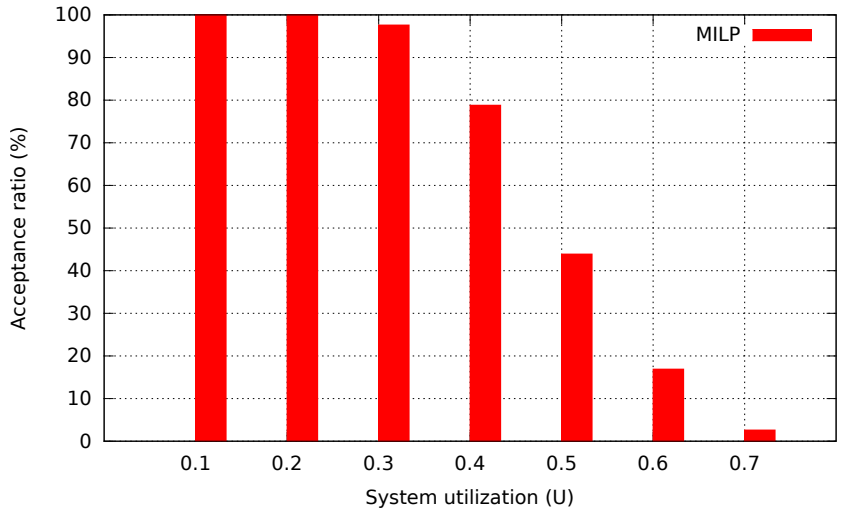


Fig. 7.14 Partial solutions: percentage of tasks that can be scheduled on the same processor. The RTSDFG size $|\mathcal{T}|$ was fixed to 30 tasks.

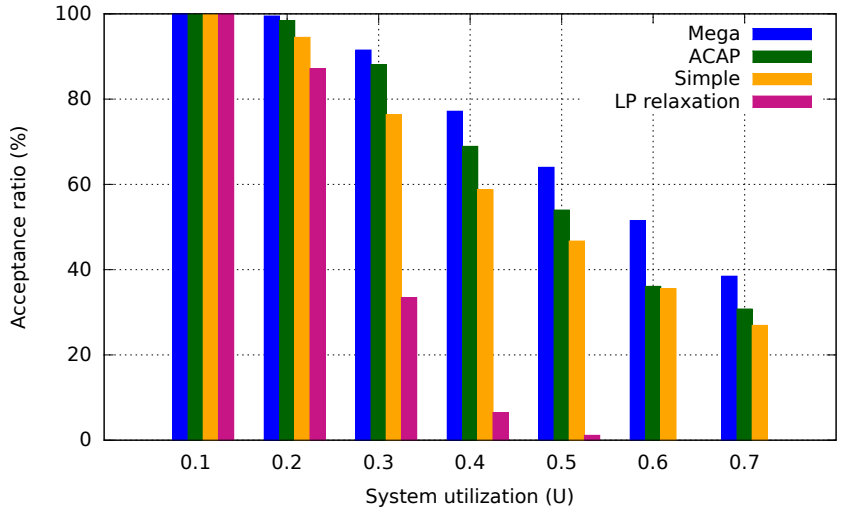
Flexible **case**

In this experiment, we generated harmonic instances with release dates. The system utilization was varied between 0.1 and 0.7. The RTSDFG size was fixed to 30 tasks. Using the exact method for the *flexible* case, instances are sorted into feasible, infeasible and solver failure. Figure 7.15a presents the MILP acceptance ratio for the *flexible* case. Figure 7.15b gives the heuristics acceptance ratios on feasible instances. We notice that the mega-heuristic has a higher acceptance ratio compared to those of the LP relaxation, *simple* and ACAP heuristics. On the other hand, we see that the ACAP heuristic has a highest acceptance ratio compared to those of the LP relaxation and *simple* heuristics. This is due to the additional priority which schedules the tasks as close as possible. In addition, we notice that the LP relaxation acceptance ratio decreases in order to reach 6.47% and 1.14% when the system utilization is respectively equal to 0.4 and 0.5. Therefore, we deduce that the efficiency of this method is limited to a system utilization less or equal than 0.3. For all the heuristics, we notice that

the acceptance ratios decrease with the incrementation of the system utilisation. For example, the Mega acceptance ratio decreases from 100.0% for $U = 0.1$ to 38.46% for $U = 0.7$. This is mainly due to the increase of the tasks worst case execution times.



(a) MILP acceptance ratio for the *flexible* case.



(b) Heuristics acceptance ratio on feasible instances.

Fig. 7.15 Experimental results on harmonic instances with release dates. The RTSDFG size $|\mathcal{T}|$ was fixed to 30 tasks.

Figure 7.16 gives the average computation time to find a feasible schedule using different methods. Similar to the *fixed* case, we notice that our methods have a similar changing tendency i.e, their average computation times increase with the increase of the system utilization. This is due to the amount of computation that grows with the

incrementation of the tasks worst execution times. When the system utilization is less than or equal to 0.2, we see that the MILP average computation time is slightly lower than that of Mega, ACAP and LP relaxation. However, when the system utilization is greater than or equal to 0.3, our heuristics have computation times considerably lower than the exact method (MILP). For example, when the system utilization is 0.5, the average computation times of the exact method, Mega, ACAP, *simple* and LP relaxation heuristics are respectively equal to 58.93, 1.96, 2.14, 0.18 and 1.81 seconds. Hence, we deduce that the Mega heuristic is 109.12 times faster than the exact method (MILP) with an acceptance ratio of 64.01%.

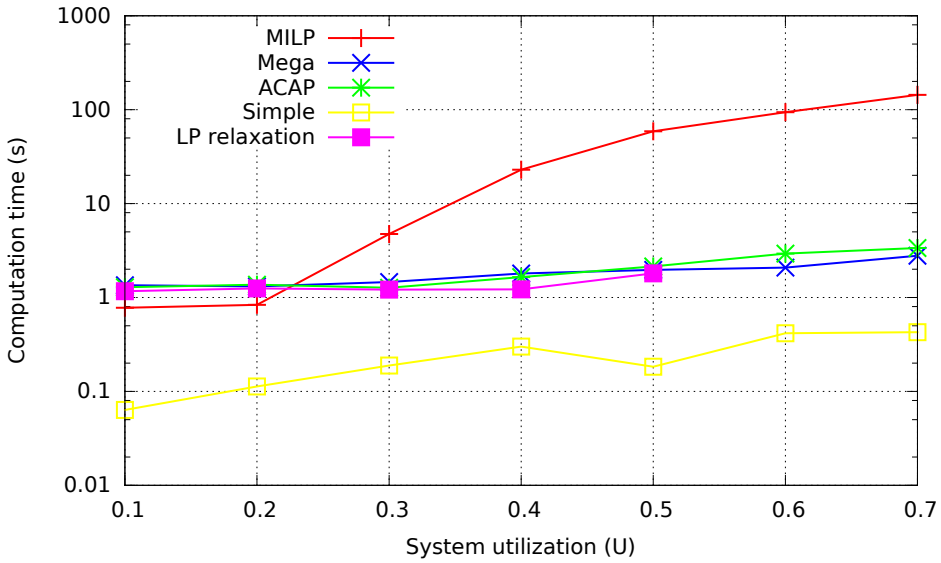


Fig. 7.16 Average computation time to find a feasible solution. MILP represents our exact method while Mega, ACAP, *simple* and LP relaxation represent our heuristics.

Comparison

In order to compare the performances of our heuristics in both *fixed* and *flexible* cases, we generated harmonic instances with and without release dates. The graph size was fixed to 30 tasks and the system utilization was varied between 0.1 and 0.8. Unlike the previous experiments, we applied the mega-heuristic for *fixed* and *flexible* cases on all the instances without any exception. The histogram in Figure 7.17 depicts the mega-heuristic acceptance ratios on harmonic instances without release dates. We notice that the acceptance ratios, in both *fixed* and *flexible* cases, decrease with the incrementation of the system utilization. We also notice that the acceptance ratio for

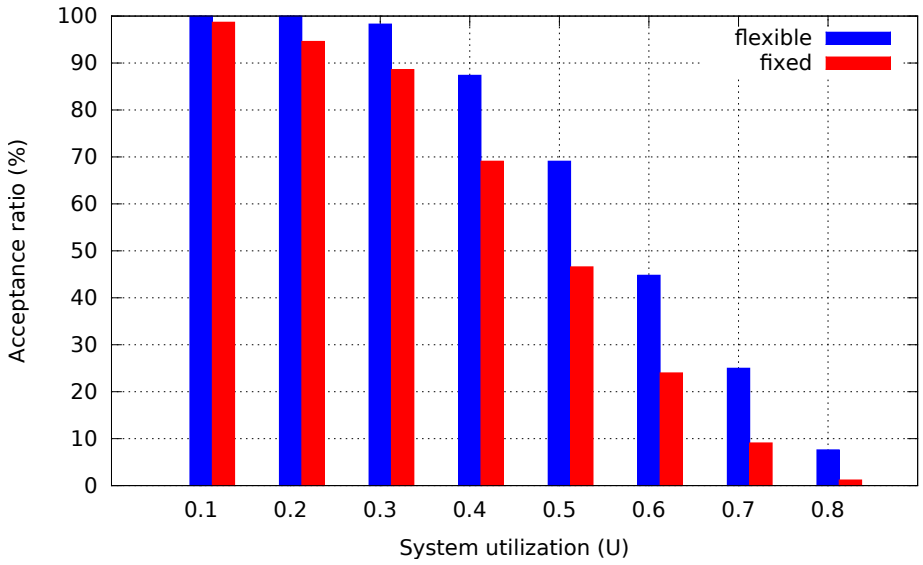


Fig. 7.17 The mega-heuristic acceptance ratio on harmonic instances without release dates. The RTSDFG size $|\mathcal{T}|$ was fixed to 30 tasks.

the *flexible* case is higher than that of the *fixed* case. Moreover, the gap between these ratios varies between 2.3% and 22.5%.

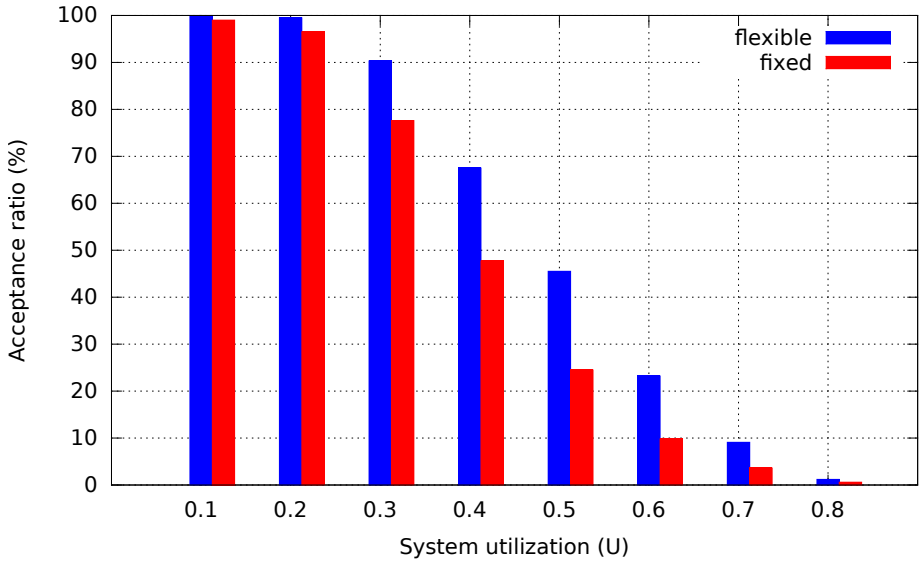


Fig. 7.18 The mega-heuristic acceptance ratio on harmonic instances with release dates. The RTSDFG size $|\mathcal{T}|$ was fixed to 30 tasks.

On the other hand, Figure 7.18 shows the mega-heuristic acceptance ratios on harmonic instances with release dates. Same as the previous results, the acceptance ratio for the *flexible* case is higher than that of the *fixed* case. Moreover, the gap between these ratios varies between 1.1% and 22.5%. Therefore, we can deduce that applying the mega-heuristic for the *flexible* case increases the possibility of finding a feasible schedule. In addition, we notice that, in both *fixed* and *flexible* cases, the mega-heuristic acceptance ratios on instances without release dates are higher than the acceptance ratios on instances with release dates. The gap between the acceptance ratios on instances with/without release dates varies between 0.5% and 23.6% in the *flexible* case. It varies between 0.3% and 22.1% in the *fixed* case. Hence, we can deduce that using our heuristics to schedule a set of tasks released at the same time is less restrictive than scheduling the same set of tasks with random release dates.

7.4 Conclusion

This chapter presented the experimental results of this thesis. It consists of two parts: latency evaluation and mono-processor scheduling.

In the first part, we evaluated the exact pricing algorithm and the upper and lower bound algorithms developed in Chapter 5. These methods compute the worst case system latency using the RTSDFG formalism. For the exact pricing algorithm, numerical experiments have shown that the average computation time to compute the worst case system latency increases by increasing the average repetition factor. However, this average computation time remains constant for the upper and lower bounds evaluation. We also found that the gap between the exact value and its upper bound is between 10 and 15%. Moreover, the gap between the exact value and the lower bound varies between 20 and 30%.

In the second part, we exposed experimentations to evaluate the performance of our exact algorithms (MILP), LP relaxation, *simple*, ACAP heuristics developed in chapter 6. These algorithms are dedicated to solve the mono-processor scheduling problem of strictly periodic system subject to communication constraints. For the exact algorithms (*fixed* and *flexible*), numerical experiments have shown that the MILP acceptance ratio increases with the incrementation of the graph size. However, this acceptance ratio decreases with the incrementation of the system utilization. Moreover, their average computation times to find a feasible solution increase exponentially according to the graph size and the system utilization. These algorithms were applied on different types

of instances in order to evaluate the impact of several parameters, such as the period ratio and the graph topology.

On the other hand, we proposed a mega-heuristic that combines several heuristics for both *fixed* and *flexible* cases. Heuristics were applied on several types of instances. Experimental results showed that the average computation time to find a feasible schedule using the mega-heuristic is much less compared to the exact method, especially when the system utilization is greater than or equal to 0.5. On the other hand, these experiments demonstrate that the mega-heuristic acceptance ratio on solver failure instances varies between 100% and 21.01% with an average computation time in the range of seconds. In addition, we found that the mega-heuristic is able to allocate on the same processor 96% to 86% of the tasks when it is not able to provide a feasible schedule. Finally, we compared the mega-heuristic for both *fixed* and *flexible* cases. We found out that the mega-heuristic acceptance ratio for the *flexible* case is greater than the *fixed* case with a gap which may reach 26.3%.

Chapter 8

Conclusion and Perspectives

Conclusion

In recent decades, electronic embedded systems have revolutionized our societies in daily lives. The increase in the application demands and the fast improvement in platform architectures led to more and more complex embedded systems. In order to ensure the optimal use of resources, several applications from different computing domains are simultaneously executed on the same platform. Advanced Driver Assistance Systems (ADAS) [117, 66] are typical examples of these applications. Their purpose is to assist drivers in making safety-related decisions, provide signals in possibly dangerous driving situations, and execute counteractive measures. These applications gather best-effort jobs (e.g. image processing for analyzing complex scenes) and critical real-time tasks (e.g. emergency braking). Therefore, this type of applications require a research effort in order to be efficiently modeled and executed. In this thesis, our study was focused on modeling and scheduling critical real-time systems using data flow formalisms.

In the first part, we defined a general and intuitive communication model within multi-periodic systems. We demonstrated that the communications between multi-periodic tasks can be directly expressed as a RTSDFG. This modeling allows precise definition of the system latency. Accordingly, we developed an exact evaluation method to compute the worst case system latency from a given input to a connected outcome. Then, we framed this value using two algorithms that compute its upper and lower bounds respectively. Finally, our experimental results show that these bounds can be computed using a polynomial amount of time, while the time required to compute the exact value increases linearly with the average repetition factor. We also found that the gap between the exact value and its upper bound varied between 10 and 15%.

Moreover, the gap between the exact value and the lower bound varied between 20 and 30%.

In the second part, we addressed the mono-processor scheduling problem of non-preemtive strictly periodic systems subject to communication constraints. Based on the RTSDFG model, we proved that scheduling two strictly periodic communicating tasks is equivalent to scheduling two independent tasks. These tasks can only be executed between their release dates and their deadlines. Accordingly, we proposed an optimal algorithm using MILP formulations. Two cases were treated: the *fixed* and *flexible* interval cases. Since the scheduling problem is know to be NP-complete in the strong sense [79], we proposed three heuristics: linear programming relaxation, *simple* and ACAP heuristics. For the second and the third heuristic if no feasible solution is found, a partial solution is computed. In comparison to the optimal algorithms, heuristics were characterized by small average computation times. The heuristics acceptance ratios on feasible instances varied between 100% and 38.46%. Moreover, their acceptance ratios on solver failure instances varied between 100% and 21.01% with an average computation time in the range of seconds. Finally, these heuristics were able to allocate on the same processor 96% to 86% of the tasks when they were not able to provide a feasible schedule.

Perspectives

This section presents perspectives and potential future work in the continuity of this thesis.

Multi/Many-processor scheduling

In this thesis, we proposed several heuristics (Mega, ACAP and *simple*) in order to solve the mono-processor scheduling problem of non-preemptive strictly periodic systems subject to communication constraints. These heuristics are much less time-consuming than the exact methods. In addition, they are able to provide a partial schedule when a feasible schedule is not found. Therefore, one of the perspectives of our work is mapping and scheduling an application modeled by a RTSDFG on a distributed architecture such as Kalray-MPPA [3]. This future work can investigate the multi/many-processor scheduling problem while taking into account the inter-processor communications. In

fact, our communication model allows us to compute, using a closed formula, the time required to transfer data between the tasks' executions.

Application: WATERS Industrial Challenge

Hamann et al. [57] proposed an automotive embedded application for WATERS Industrial Challenge 2017. This application is an engine management system which consists of several functional components. These latter are tightly related by communication dependencies and data sharing, while having real-time requirements. This application is composed of:

- 1250 runnables which represent pieces of codes that read and write a set of labels.
- 10000 labels that symbolize the communication variables between runnables. Producing (writing) a label is restricted to at most one runnable. However, consuming (reading) a label can be done by several runnables.
- 21 tasks which group several runnables. Periodic tasks gather several runnables that have the same period, while sporadic tasks assemble runnables that have the same minimal and maximal inter arrival times.

Logical Execution Time (LET) has been proposed to ensures temporal determinism by decoupling computation and communication. In this context, it would be interesting to model this application using the RTSDFG formalism, since the communication model behind this formalism is similar to the LET concept.

Relax the strict periodicity constraint

In this thesis, we addressed the scheduling problem of non-preemptive strictly periodic systems subject to communication constraints. The strict periodicity allowed us to define the resource constraints with a polynomial number of constraints on the starting dates of the tasks' first executions. Accordingly, we developed an exact method and several heuristics that solve the scheduling problem. An interesting perspective is to address the non-preemptive scheduling problem by relaxing the strict periodicity constraint. This problem can be solved using the list scheduling algorithms [53, 81]. In fact, it would be worth comparing our results, in terms of acceptance ratio and average computation time, to the results obtained by relaxing the strict periodicity constraint.

References

[1] Al Sheikh, A., Brun, O., Hladik, P.-E., and Prabhu, B. J. (2011). A best-response algorithm for multiprocessor periodic scheduling. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 228–237. IEEE.

[2] Ali, H. I., Akesson, B., and Pinho, L. M. (2015). Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 701–710. IEEE.

[3] Aubry, P., Beaucamps, P.-E., Blanc, F., Bodin, B., Carpov, S., Cudennec, L., David, V., Doré, P., Dubrulle, P., De Dinechin, B. D., et al. (2013). Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. *Procedia Computer Science*, 18:1624–1633.

[4] Baccelli, F., Cohen, G., Olsder, G. J., and Quadrat, J.-P. (1992). Synchronization and linearity: an algebra for discrete event systems.

[5] Bamakhrama, M. and Stefanov, T. (2011). Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 195–204. ACM.

[6] Bamakhrama, M. A. and Stefanov, T. (2012). Managing latency in embedded streaming applications under hard-real-time scheduling. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 83–92. ACM.

[7] Bamakhrama, M. A., Zhai, J. T., Nikolov, H., and Stefanov, T. (2012). A methodology for automated design of hard-real-time embedded streaming systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 941–946. EDA Consortium.

[8] Baruah, S. (2007). Techniques for multiprocessor global schedulability analysis. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 119–128. IEEE.

[9] Baruah, S. K. (2006). The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 32(1):9–20.

[10] Baruah, S. K., Mok, A. K., and Rosier, L. E. (1990a). Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190. IEEE.

[11] Baruah, S. K., Rosier, L. E., and Howell, R. R. (1990b). Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-time systems*, 2(4):301–324.

[12] Benabid-Najjar, A., Hanen, C., Marchetti, O., and Munier-Kordon, A. (2012). Periodic schedules for bounded timed weighted event graphs. *IEEE Transactions on Automatic Control*, 57(5):1222–1232.

[13] Benveniste, A., Le Guernic, P., and Jacquemot, C. (1991). Synchronous programming with events and relations: the signal language and its semantics. *Science of computer programming*, 16(2):103–149.

[14] Bernat, G., Burns, A., and Liamosi, A. (2001). Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321.

[15] Berry, G. (1989). *Real time programming: Special purpose or general purpose languages*. PhD thesis, INRIA.

[16] Berry, G. and Gonthier, G. (1992). The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152.

[17] Bilsen, G., Engels, M., Lauwereins, R., and Peperstraete, J. (1996). Cycle-static dataflow. *IEEE Transactions on signal processing*, 44(2):397–408.

[18] Bini, E. and Buttazzo, G. C. (2005). Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154.

[19] Bodin, B., Lesparre, Y., Delosme, J.-M., and Munier-Kordon, A. (2014). Fast and efficient dataflow graph generation. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, pages 40–49. ACM.

[20] Bouakaz, A., Talpin, J.-P., and Vitek, J. (2012). Affine data-flow graphs for the synthesis of hard real-time applications. In *Application of Concurrency to System Design (ACSD), 2012 12th International Conference on*, pages 183–192. IEEE.

[21] Brandes, U. (2001). A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177.

[22] Burns, A. and Wellings, A. J. (2001). *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education.

[23] Caspi, P., Mazuet, C., and Paligot, N. (2001). About the design of distributed control systems: The quasi-synchronous approach. *Computer Safety, Reliability and Security*, pages 215–226.

[24] Chabrol, D., David, V., Aussagues, C., Louise, S., and Daumas, F. (2005). Deterministic distributed safety-critical real-time systems within the oasis approach. In *IASTED PDCS*, pages 260–268.

[25] Chen, J., Du, C., Xie, F., and Lin, B. (2016a). Allocation and scheduling of strictly periodic tasks in multi-core real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2016 IEEE 22nd International Conference on*, pages 130–138. IEEE.

[26] Chen, J., Du, C., Xie, F., and Yang, Z. (2016b). Schedulability analysis of non-preemptive strictly periodic tasks in multi-core real-time systems. *Real-Time Systems*, 52(3):239–271.

[27] Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., and Pouzet, M. (2006). N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. *ACM SIGPLAN Notices*, 41(1):180–193.

[28] Cohen, G., Dubois, D., Quadrat, J., and Viot, M. (1985). A linear-system-theoretic view of discrete-event processes and its use for performance evaluation in manufacturing. *IEEE transactions on Automatic Control*, 30(3):210–220.

[29] Cucu, L. and Sorel, Y. (2003). Schedulability condition for systems with precedence and periodicity constraints without preemption. In *Proceedings of 11th Real-Time Systems Conference, RTS'03*.

[30] Cucu, L. and Sorel, Y. (2004). Non-preemptive multiprocessor scheduling for strict periodic systems with precedence constraints.

[31] Dantzig, G. B. and Wolfe, P. (1960). Decomposition principle for linear programs. *Operations research*, 8(1):101–111.

[32] Davis, A. L. and Keller, R. M. (1982). Data flow program graphs.

[33] Davis, R. I. and Burns, A. (2011). A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):35.

[34] Dennis, J. B. (1974). First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer.

[35] Do, X. K. (2016). *Modèle de calcul et d'exécution pour des applications flots de données dynamiques avec contraintes temps réel*. PhD thesis, Paris 6.

[36] Do, X. K., Louise, S., and Cohen, A. (2015). Managing the latency of data-dependent tasks in embedded streaming applications. In *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*, pages 9–16. IEEE.

[37] Eisenbrand, F., Hähnle, N., Niemeier, M., Skutella, M., Verschae, J., and Wiese, A. (2010a). Scheduling periodic tasks in a hard real-time environment. *Automata, languages and programming*, pages 299–311.

[38] Eisenbrand, F., Kesavan, K., Mattikalli, R. S., Niemeier, M., Nordsieck, A. W., Skutella, M., Verschae, J., and Wiese, A. (2010b). Solving an avionics real-time scheduling problem by advanced ip-methods. In *European Symposium on Algorithms*, pages 11–22. Springer.

[39] Elloy, J.-P. (1991). Les contraintes du temps réel dans les systèmes industriels répartis. *Revue générale de l'électricité*, (2):26–30.

[40] Feiler, P. H., Gluch, D. P., and Hudak, J. J. (2006). The architecture analysis & design language (aadl): An introduction. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

[41] Forget, J. (2009). A synchronous language for critical embedded systems with multiple real-time constraints. *Ph. D. dissertation*.

[42] Forget, J., Boniol, F., Lesens, D., and Pagetti, C. (2008). A multi-periodic synchronous data-flow language. In *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, pages 251–260. IEEE.

[43] Forget, J., Boniol, F., Lesens, D., and Pagetti, C. (2010). A real-time architecture design language for multi-rate embedded control systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 527–534. ACM.

[44] Fudenberg, D. and Tirole, J. (1991). Game theory. Technical report, The MIT press.

[45] George, L., Rivierre, N., and Spuri, M. (1996). *Preemptive and non-preemptive real-time uniprocessor scheduling*. PhD thesis, Inria.

[46] Gilles, K. (1974). The semantics of a simple language for parallel programming. *Information processing*, 74:471–475.

[47] Girault, A., Kalla, H., and Sorel, Y. (2003). Une heuristique d'ordonnancement et de distribution tolérante aux pannes pour systemes temps-réel embarqués. *Modélisation des Systemes Réactifs, MSR'03*, pages 145–160.

[48] Goddard, S. (1997). Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 60–71. IEEE.

[49] Goossens, J. (2003). Scheduling of offset free systems. *Real-Time Systems*, 24(2):239–258.

[50] Goossens, J. and Macq, C. (2001). Limitation of the hyper-period in real-time periodic task set generation. In *In Proceedings of the RTS Embedded System (RTS'01*. Citeseer.

[51] Goossens, K., Azevedo, A., Chandrasekar, K., Gomony, M. D., Goossens, S., Koedam, M., Li, Y., Mirzoyan, D., Molnos, A., Nejad, A. B., et al. (2013). Virtual execution platforms for mixed-time-criticality systems: the compsoc architecture and design flow. *ACM SIGBED Review*, 10(3):23–34.

[52] Goubier, T., Sirdey, R., Louise, S., and David, V. (2011). σc: A programming model and language for embedded manycores. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 385–394. Springer.

[53] Graham, R. L. (1966). Bounds for certain multiprocessing anomalies. *Bell Labs Technical Journal*, 45(9):1563–1581.

[54] Grandpierre, T., Lavarenne, C., and Sorel, Y. (1999). Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of the seventh international workshop on Hardware/software codesign*, pages 74–78. ACM.

[55] Guan, N., Yi, W., Deng, Q., Gu, Z., and Yu, G. (2011). Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *Journal of Systems Architecture*, 57(5):536–546.

[56] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320.

[57] Hamann, A., Dasari, D., Kramer, S., Pressler, M., Wurst, F., and Ziegenbein, D. (2017). Waters industrial challenge 2017.

[58] Harel, D. and Pnueli, A. (1985). *On the development of reactive systems*. Weizmann Institute of Science. Department of Applied Mathematics.

[59] Hausmans, J. P., Geuns, S. J., Wiggers, M. H., and Bekooij, M. J. (2014). Temporal analysis flow based on an enabling rate characterization for multi-rate applications executed on mpsocs with non-starvation-free schedulers. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, pages 108–117. ACM.

[60] Hausmans, J. P., Wiggers, M. H., Geuns, S. J., and Bekooij, M. J. (2013). Dataflow analysis for multiprocessor systems with non-starvation-free schedulers. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, pages 13–22. ACM.

[61] Henzinger, T. A., Horowitz, B., and Kirsch, C. M. (2003a). Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99.

[62] Henzinger, T. A., Horowitz, B., and Kirsch, C. M. (2003b). Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99.

[63] Hoffman, K. and Padberg, M. (1985). Lp-based combinatorial problem solving. In *Computational Mathematical Programming*, pages 55–123. Springer.

[64] Hyman, J. M., Lazar, A. A., and Pacifici, G. (1991). Real-time scheduling with quality of service constraints. *IEEE Journal on Selected Areas in Communications*, 9(7):1052–1063.

[65] Jeffay, K., Stanat, D. F., and Martel, C. U. (1991). On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139. IEEE.

[66] Jones, W. D. (2002). Building safer cars. *IEEE Spectrum*, 39(1):82–85.

[67] Jonkman, J. M. and Buhl Jr, M. L. (2005). Fast user's guide-updated august 2005. Technical report, National Renewable Energy Laboratory (NREL), Golden, CO.

[68] Kantorovich, L. V. (1960). Mathematical methods of organizing and planning production. *Management Science*, 6(4):366–422.

[69] Karp, R. M. and Miller, R. E. (1966). Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411.

[70] Kermia, O., Cucu, L., and Sorel, Y. (2006). Non-preemptive multiprocessor static scheduling for systems with precedence and strict periodicity constraints. In *Proceedings of the 10th International Workshop On Project Management and Scheduling, PMS'06*.

[71] Kermia, O. and Sorel, Y. (2007). A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor. In *Proceedings of ISCA 20th international conference on Parallel and Distributed Computing Systems, PDCS'07*.

[72] Kermia, O. and Sorel, Y. (2008). Schedulability analysis for non-preemptive tasks under strict periodicity constraints. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on*, pages 25–32. IEEE.

[73] Khatib, J., Munier-Kordon, A., Klikpo, E. C., and Trabelsi-Colibet, K. (2016). Computing latency of a real-time system modeled by synchronous dataflow graph. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 87–96. ACM.

[74] Kirsch, C. M. (2002). Principles of real-time programming. In *EMSOFT*, volume 2, pages 61–75. Springer.

[75] Kirsch, C. M., Sanvido, M. A., Henzinger, T. A., and Pree, W. (2002). A giotto-based helicopter control system. In *International Workshop on Embedded Software*, pages 46–60. Springer.

[76] Klikpo, E. C., Khatib, J., and Munier-Kordon, A. (2016). Modeling multi-periodic simulink systems by synchronous dataflow graphs. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*, pages 1–10. IEEE.

[77] Kopetz, H. (2002). Time-triggered real-time computing. *IFAC Proceedings Volumes*, 35(1):59–70.

[78] Kopetz, H. (2011). *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media.

[79] Korst, J., Aarts, E., and Lenstra, J. K. (1996). Scheduling periodic tasks. *INFORMS journal on Computing*, 8(4):428–435.

[80] Korst, J., Aarts, E., Lenstra, J. K., and Wessels, J. (1991). Periodic multiprocessor scheduling. In *PARLE'91 Parallel Architectures and Languages Europe*, pages 166–178. Springer.

[81] Kwok, Y.-K. and Ahmad, I. (1999). Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422.

[82] Land, A. H. and Doig, A. G. (2010). An automatic method for solving discrete programming problems. *50 Years of Integer Programming 1958-2008*, pages 105–132.

[83] Lee, E. and Messerschmitt, D. (1987a). Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245.

[84] Lee, E. A. and Messerschmitt, D. G. (1987b). Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35.

[85] Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61.

[86] Liu, D., Spasic, J., Zhai, J. T., Stefanov, T., and Chen, G. (2014). Resource optimization for csdf-modeled streaming applications with latency constraints. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE.

[87] Louise, S., David, V., Delcoigne, J., and Aussagues, C. (2002). Oasis project: deterministic real-time for safety critical embedded systems. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 223–226. ACM.

[88] Magnanti, T. L. and Vachani, R. (1990). A strong cutting plane algorithm for production scheduling with changeover costs. *Operations Research*, 38(3):456–473.

[89] Mandel, L., Plateau, F., and Pouzet, M. (2010). Lucy-n: a n-synchronous extension of lustre. In *MPC*, volume 6120, pages 288–309. Springer.

[90] Marchetti, O. and Kordon, A. M. (2009). Cyclic scheduling for the synthesis of embedded systems. *Introduction to scheduling*, pages 135–164.

[91] Marchetti, O. and Munier-Kordon, A. (2009). A sufficient condition for the liveness of weighted event graphs. *European Journal of Operational Research*, 197(2):532–540.

[92] Marouf, M. and Sorel, Y. (2010). Schedulability conditions for non-preemptive hard real-time tasks with strict period. In *18th International Conference on Real-Time and Network Systems RTNS'10*.

[93] Marouf, M. and Sorel, Y. (2011). Scheduling non-preemptive hard real-time tasks with strict periods. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–8. IEEE.

[94] Mok, A. K.-L. (1983). *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology.

[95] Moreira, O. M. and Bekooij, M. J. (2007). Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007(1):083710.

[96] Munier, A. (1993). Optimal asymptotic behavior of a generalized timed event graph: application to an assembly system. *APII*, 27:487–487.

[97] Nasri, M., Baruah, S., Fohler, G., and Kargahi, M. (2014). On the optimality of rm and edf for non-preemptive real-time harmonic tasks. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 331. ACM.

[98] Nasri, M. and Kargahi, M. (2014). Precautious-rm: a predictable non-preemptive scheduling algorithm for harmonic tasks. *Real-Time Systems*, 50(4):548–584.

[99] Park, C. and Shaw, A. C. (1990). Experiments with a program timing tool based on source-level timing schema. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 72–81. IEEE.

[100] Park, M. (2007). Non-preemptive fixed priority scheduling of hard real-time periodic tasks. *Computational Science–ICCS 2007*, pages 881–888.

[101] Piaggio, M., Sgorbissa, A., and Zaccaria, R. (2000). Pre-emptive versus non-pre-emptive real time scheduling in intelligent mobile robotics. *Journal of Experimental & Theoretical Artificial Intelligence*, 12(2):235–245.

[102] Pira, C. and Artigues, C. (2013). Propagation mechanism for a non-preemptive strictly periodic scheduling problem.

[103] Pouzet, M. (2006). Lucid synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI*.

[104] Puschner, P. and Koza, C. (1989). Calculating the maximum execution time of real-time programs. *Real-time systems*, 1(2):159–176.

[105] Savelsbergh, M. (1997). A branch-and-price algorithm for the generalized assignment problem. *Operations research*, 45(6):831–841.

[106] Shaw, A. C. (1989). Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889.

[107] Sifakis, J., Tripakis, S., and Yovine, S. (2003). Building models of real-time systems from application software. *Proceedings of the IEEE*, 91(1):100–111.

[108] Sih, G. C. and Lee, E. A. (1990). Scheduling to account for interprocessor communication within interconnection-constrained processor networks. In *ICPP (1)*, pages 9–16.

[109] Singh, A., Ekberg, P., and Baruah, S. (2017). Applying real-time scheduling theory to the synchronous data flow model of computation. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[110] Sprunt, B., Sha, L., and Lehoczky, J. (1989). Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60.

[111] Stankovic, J. A. (1988). Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19.

[112] Stuijk, S., Geilen, M., and Basten, T. (2006). Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 899–904. IEEE.

[113] Tabuada, P. (2007). Event-triggered real-time scheduling of stabilizing control tasks. *IEEE Transactions on Automatic Control*, 52(9):1680–1685.

[114] Tendulkar, P., Poplavko, P., and Maler, O. (2014). Strictly periodic scheduling of acyclic synchronous dataflow graphs using smt solvers. Technical report, Technical report TR-2014-5. Verimag research report.

[115] Thies, W., Karczmarek, M., and Amarasinghe, S. (2002a). Streamit: A language for streaming applications. In *Compiler Construction*, pages 49–84. Springer.

[116] Thies, W., Lin, J., and Amarasinghe, S. (2002b). Phased computation graphs in the polyhedral model.

[117] Vlacic, L., Parent, M., and Harashima, F. (2001). *Intelligent vehicle technologies*. Butterworth-Heinemann.

[118] Wirth, N. (1977). Toward a discipline of real-time programming. *Communications of the ACM*, 20(8):577–583.