Real-time scheduling of dataflow graphs

Adnan Bouakaz

▶ **To cite this version:**

Adnan Bouakaz. Real-time scheduling of dataflow graphs. Other [cs.OH]. Université de Rennes, 2013. English. NNT : 2013REN1S103 . tel-00945453

HAL Id: tel-00945453
https://theses.hal.science/tel-00945453

Submitted on 12 Feb 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ANNÉE 2013

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Adnan BOUAKAZ

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Systèmes Aléatoires
ISTIC

Real-time scheduling of dataflow graphs

Thèse soutenue à Rennes
le 27 novembre 2013

devant le jury composé de :

Isabelle PUAUT
Professeur à l'Université de Rennes 1 / *Présidente*

Robert DE SIMONE
Directeur de recherche INRIA / *Rapporteur*

Sander STUIJK
Prof. adjoint à Eindhoven University of Tech. / *Rapporteur*

Frank SINGHOFF
Professeur à l'Université de Brest / *Examinateur*

Eric JENN
Responsable de projet à Thalès Avionique Toulouse / *Examinateur*

Jean-Pierre TALPIN
Directeur de recherche INRIA / *Directeur de thèse*

Jan VITEK
Professeur à Purdue University / *Co-directeur de thèse*

Acknowledgements

First, I would like to greatly thank all the members of my dissertation committee. I wish to thank Isabelle PUAUT, professor of University of Rennes 1, for her acceptance to be president of the committee and for being my master's advisor.

I would like to thank Robert de SIMONE, INRIA research director, and Sander Stuijk, assistant professor of Eindhoven University of Technology, for accepting to review and evaluate this thesis. I am also very thankful for Frank SINGHOFF, professor of University of Bretagne Occidentale, and Eric JENN, project manager of THALES Avionics, for accepting to be the examiners of my thesis defense.

This thesis would not be possible without the guidance of my thesis advisors, Jean-Pierre TALPIN, INRIA research director, and Jan VITEK, professor of Purdue University. I would like to thank them for accepting me as a Ph.D. student, for their continuous interest and encouragements, and for always pushing for results. I wish to thank Thierry GAUTIER, INRIA researcher, for his help preparing the final thesis manuscript and presentation.

I would like to thank my colleagues within the ESPRESSO team at IRISA laboratory for sharing the good ambiance during my stay at IRISA. I wish to thank Alěs Plšek for his contribution to my successful visit to Purdue University.

Finally, I am deeply thankful to my parents for their endless love and support, to my brothers, sisters, and their families. I wish to thank Mohamed-Elarbi DJEBBAR, for his support and encouragement, and all my friends at Rennes, among others, Abdallah, Mehdi, Rabie, Youcef, Mohammed, Mouaad, Nadjib, and Hamza.

to my parents . . .

Contents

Introduction **3**

Résumé en français **7**

1 Design and Verification **11**

1.1 Generalities . . . 11
1.2 Dataflow models of computation . . . 13
1.2.1 Kahn process networks . . . 14
1.2.2 Bounded execution of KPNs . . . 15
1.2.3 Dataflow process networks . . . 17
1.2.4 Specific dataflow graph models . . . 18
1.2.5 Dataflow synchronous model . . . 21
1.3 Static analysis of (C|H)SDF graphs . . . 26
1.3.1 Reachability analysis . . . 26
1.3.2 Timing analysis . . . 29
1.3.3 Memory analysis . . . 33
1.4 Real-time scheduling . . . 35
1.4.1 System models and terminology . . . 36
1.4.2 EDF schedulability analysis . . . 39
1.4.3 Fixed-priority schedulability analysis . . . 43
1.4.4 Symbolic schedulability analysis . . . 45
1.4.5 Real-time scheduling of dataflow graphs . . . 47
1.5 Real-time calculus . . . 48
1.6 Conclusion . . . 49

2 Abstract schedules **51**

2.1 Priority-driven operational semantics . . . 52
2.2 Activation-related schedules . . . 53
2.2.1 Activation relations . . . 53
2.2.2 Consistency . . . 55
2.2.3 Overflow analysis . . . 63
2.2.4 Underflow analysis . . . 65
2.3 Affine schedules . . . 67
2.3.1 Affine relations . . . 68

2.3.2 Consistency . . . 70
2.3.3 Fixed-priority schedules . . . 72
2.3.4 EDF schedules . . . 76
2.4 Specific cases . . . 78
2.4.1 Ultimately cyclo-static dataflow graphs . . . 79
2.4.2 Multichannels . . . 81
2.4.3 Shared storage space . . . 82
2.4.4 FRStream . . . 83
2.5 Conclusion . . . 86

3 Symbolic schedulability analysis 87
3.1 General conditions . . . 87
3.2 Fixed-priority scheduling . . . 90
3.2.1 Priority assignment . . . 91
3.2.2 Uniprocessor scheduling . . . 97
3.2.3 Multiprocessor scheduling . . . 101
3.3 EDF scheduling . . . 102
3.3.1 Deadlines adjustment . . . 102
3.3.2 Uniprocessor scheduling . . . 106
3.3.3 Multiprocessor scheduling . . . 109
3.4 Conclusion . . . 112

4 Experimental validation 113
4.1 Performance comparison: ADFG vs DARTS . . . 113
4.1.1 Throughput . . . 115
4.1.2 Buffering requirements . . . 118
4.2 Symbolic schedulability analysis . . . 121
4.2.1 EDF scheduling . . . 122
4.2.2 Fixed-priority scheduling . . . 125
4.3 Application: Design of SCJ/L1 systems . . . 127
4.3.1 Concurrency model of SCJ/L1 . . . 128
4.3.2 Dataflow design model . . . 129
4.4 Conclusion . . . 131

Conclusion 133

Bibliography 150

A Sets, orders, and sequences 151

List of Figures 157

Introduction

Embedded systems are everywhere: homes, cars, airplanes, phones, etc. Every time we take a picture, watch TV, or answer the phone, we are interacting with an embedded system. The number of embedded systems in our daily lives is growing ceaselessly. A modern heart pacemaker is an example of so-called *safety-critical* embedded systems, a term that refers to systems whose failure might endanger human life or cause an unacceptable damage. A heart pacemaker is also a *real-time* embedded system, a term that refers to systems which must perform their functions in time frames dictated by the environment.

Embedded systems design requires approaches taking into account non-functional requirements regarding optimal use of resources (e.g. memory, energy, time, etc.) while ensuring safety and robustness. Therefore, design approaches should use as much as possible formal models of computation to describe the behavior of the system, formal analysis to check safety properties, and automatic synthesis to produce "correct by construction" implementations. Many models of computation for embedded systems design have been proposed in the past decades. Most of them have to deal with *time* and *concurrency*. Among these models, the dataflow and the periodic task models are very popular.

Dataflow models of computation Dataflow models are characterized by a data-driven style of control; data are processed while flowing through a network of computation nodes. There are three major variants of dataflow models in the literature, namely, Kahn process networks, dataflow process networks, and dataflow synchronous languages. These models are widely used to design *stream-based* systems. The *Turing-completeness* of Kahn and dataflow process networks has motivated the research community to propose less expressive but decidable dataflow models such as synchronous and cyclo-static dataflow graphs. In these models, a system is described by a directed graph where computation nodes (called actors) communicate only through one-to-one FIFO buffers. Each time an actor executes (or fires), it consumes a predefined (either constant or cyclically changing) number of tokens from its input channels and produces a predefined number of tokens on its output channels. These two models are used in the signal and stream processing community due to the following reasons:

Expressiveness: They are expressive enough to model most of digital signal processing algorithms (e.g. FFT, H.263 decoder, etc.).

Decidability : The questions of *boundedness* (i.e. whether the system can be executed with bounded buffers) and *liveness* (i.e. whether each actor can fire infinitely often) are decidable. Furthermore, static-periodic schedules (i.e. infinite repetitions of firing sequences of actors) can be easily constructed.

Powerful tools : Many tools (e.g. SDF^3, *Ptolemy*) propose algorithms for static-periodic scheduling of dataflow graphs considering many optimization problems: buffer minimization, throughput maximization, code size minimization, etc.

Real-time scheduling The major drawback of static-periodic schedules (or cyclic executives) is their inflexibility and difficult maintainability. Therefore, new scheduling techniques have been proposed; e.g. fixed-priority scheduling, earliest-deadline first scheduling, etc. Their underlying model of computation consists of a set of (periodic, sporadic, or aperiodic) independent and concurrent real-time tasks. Each task is characterized by a deadline that must be met by all invocations (called jobs) of that task. The real-time research community has proposed a set of algorithms (called *schedulability* tests) to verify before the costly implementation phase whether tasks will meet their deadlines or not.

Nowadays real-time embedded systems are so complex that real-time operating systems are used to manage hardware resources and host real-time tasks. Most of real-time operating systems implement a bunch of priority-driven scheduling algorithms (as deadline monotonic and earliest-deadline first scheduling policies). In parallel with the rapid increase in software complexity, demands for increasing processor performance have motivated using multiprocessor platforms for real-time embedded applications. While not as mature as uniprocessor scheduling theory, multiprocessor scheduling theory starts providing us with very interesting schedulability tests.

Problem statement

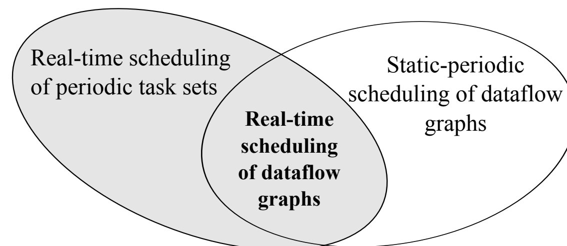


Figure 1: Real-time scheduling of dataflow graphs.

The key properties of any real-time safety-critical system are *functional determinism* (i.e. for the same sequence of inputs, the system always produces the same sequence of outputs) and *temporal predictability* (i.e. tasks meet their deadlines even in the worst-case scenario). Besides of all the above mentioned advantages of dataflow graphs, functional determinism is an *inherent* property of this model of computation. This is why we argue

that dataflow graphs are suitable for designing safety-critical systems. Thus, the input of our synthesis technique is a dataflow graph specification.

Since real-time operating systems are used in most recent real-time systems, we must ask the following question: *how to generate a set of independent real-time tasks (or precisely a periodic task set) from a dataflow specification?* As depicted in Figure 1, we refer to this problem as real-time scheduling of dataflow graphs. Many properties (from both communities) should be satisfied for correct implementations. The most important ones are boundedness, data dependencies, and schedulability on a given architecture.

Contributions

In this thesis, we propose a method to generate implementations of dataflow graph specifications on systems equipped with a real-time scheduler (as in real-time operating systems or real-time Java virtual machines). Each actor is mapped to a periodic task with the appropriate scheduling and timing parameters (e.g. periods, priorities, first start times). The approach consists of two steps:

Abstract scheduling

An abstract schedule is a set of *timeless* scheduling constraints (e.g. relation between the speeds of two actors or between their first start times, the priority ordering, etc.). The schedule must ensure the following property: communication buffers are overflow and underflow-free; i.e. an actor never attempts to read from an empty channel or write to a full one. The appropriate buffer sizes and number of initial tokens are computed in accordance with that property.

Since each actor is implemented as an independent thread, the code size minimization problem is no longer an issue. Only the buffer minimization problem is considered at this step. Furthermore, the abstract schedule construction is entirely code and machine-independent; i.e. we do not consider neither the implementation code of actors nor their worst-case execution times on the target machine. Though (theoretically speaking) time-dependent techniques are more accurate, the smallest change in execution times (either by changing the target architecture or the implementation code) requires reconstruction of the schedule. To sum up the first step, this thesis makes the following contributions:

- We propose a general framework that uses only infinite integer sequences to describe abstract priority-driven schedules. The necessary conditions for overflow/underflow-free and consistent schedules are also presented.
- Regrading real-time scheduling of dataflow graphs, a specific class of abstract schedules (called *affine schedules*) is presented together with an ILP (Integer Linear programming) formalization of the problem.
- We also present the ultimately cyclo-static dataflow model (a generalization of the cyclo-static dataflow model) and `FRStream` (a simple synchronous language).

Symbolic schedulability analysis

Each real-time task must be characterized with some timing parameters (e.g. periods, first start times, deadlines). However, abstract scheduling does not attribute any timing properties to actors. Thus, the symbolic schedulability analysis consists in defining the scheduling parameters that: respect the abstract scheduling constraints, ensure the schedulability for a given scheduling algorithm and a given architecture, and optimize a cost function (e.g. maximize the throughput, minimize the buffering requirements, minimize the energy consumption, etc.).

This thesis presents several symbolic schedulability analyses regarding the earliest-deadline first and fixed-priority scheduling policies. Both uniprocessor and homogeneous multiprocessor scheduling are considered. Though the thesis focuses more on scheduling of connected dataflow graphs, we have also presented few symbolic schedulability algorithms for disconnected dataflow graphs.

Thesis overview

This thesis is organized as follows.

Chapter 1 first introduces the three major varieties of dataflow models of computation. We only present the most important properties regarding their semantics and expressiveness. The second part briefly reviews the existing results about static-periodic scheduling of synchronous and cyclo-static dataflow graphs. Finally, the real-time scheduling theory is briefly introduced together with the very few existing results about parametric schedulability analysis. All the mathematical concepts used in this chapter are introduced in Appendix A.

Chapter 2 presents the abstract scheduling step. Again, all needed mathematical concepts and notations about sequences can be found in Appendix A. This chapter first describes activation-related schedules (i.e. the general framework) and then affine schedules with more details on the overflow and underflow analyses w.r.t. earliest-deadline first and fixed-priority scheduling policies. The chapter ends with notes about some specific cases.

Chapter 3 presents the symbolic schedulability analysis step. Two performance metrics are considered: buffer minimization and throughput maximization. Furthermore, two scheduling policies are addressed: earliest-deadline first and fixed-priority scheduling policies for both uniprocessor and multiprocessor systems.

Chapter 4 presents the results obtained by the scheduling algorithms on a set of real-life stream processing applications and randomly generated dataflow graphs w.r.t. buffer minimization and throughput maximization. It also briefly presents how to use the affine scheduling technique to automatically generate safety-critical Java Level 1 applications from a dataflow specification.

Finally, we end this thesis with some conclusions and perspectives for future work.

Résumé en français

Les systèmes embarqués sont omniprésents dans l'industrie comme dans la vie quotidienne. Ils sont qualifiés de *critiques* si une défaillance peut mettre en péril la vie humaine ou conduire à des conséquences inacceptables. Ils sont aussi qualifiés de *temps-réel* si leur correction ne dépend pas uniquement des résultats logiques mais aussi de l'instant où ces résultats ont été produits. Les méthodes de conception de tels systèmes doivent utiliser, autant que possible, des modèles formels pour représenter le comportement du système, afin d'assurer, parmi d'autres propriétés, le déterminisme fonctionnel et la prévisibilité temporelle.

Les graphes « flot de données », grâce à leur déterminisme fonctionnel *inhérent*, sont très répandus pour modéliser les systèmes embarqués de traitement de flux. Dans ce modèle de calcul, un système est décrit par un graphe orienté, où les nœuds de calcul (ou acteurs) communiquent entre eux à travers des buffers FIFO. Un acteur est activé lorsqu'il y a des données suffisantes sur ses entrées. Une fois qu'il est actionné, l'acteur consomme (resp. produit) un nombre prédéfini de données (ou jetons) à partir de ses entrées (resp. sur ses sorties). L'ordonnancement statique et périodique des graphes flot de données a été largement étudié surtout pour deux modèles particuliers : SDF et CSDF. Le problème consiste à identifier des séquences périodiques infinies d'actionnement des acteurs qui aboutissent à des exécutions *complètes* et *à buffers bornés*. Le problème est abordé sous des angles différents : maximisation de débit, minimisation des tailles des buffers, etc. Cependant, les ordonnancements statiques sont trop rigides et difficile à maintenir.

Aujourd'hui, les systèmes embarqués temps-réel sont très complexes, au point qu'ils ont recours à des systèmes d'exploitation temps-réel (RTOS) pour gérer les tâches concurrentes et les ressources critiques. La plupart des RTOS implémentent des stratégies d'ordonnancement temps-réel dynamique ; par exemple, RM, EDF, etc. Le modèle de calcul sous-jacent consiste en un ensemble de tâches périodiques (ou non) indépendantes ; chaque tâche est caractérisée par des paramètres d'ordonnancement (ex. : période, échéance, priorité, etc.). La théorie de l'ordonnancement temps-réel fournit de nombreux tests d'ordonnançabilité pour déterminer avant la phase d'implémentation, et pour une architecture et une stratégie d'ordonnancement données, si les tâches vont respecter leurs échéances.

Il est intéressant de pouvoir modéliser les systèmes embarqués par des graphes flot de données et en même temps d'être capable d'implémenter les acteurs par des tâches temps-réel indépendantes ordonnançables par un RTOS. Cette thèse aborde le problème

d'ordonnancement temps-réel dynamique des graphes flot de données. Le problème n'est pas anodin et il est résolu en deux étapes : la construction d'un ordonnancement affine abstrait et l'analyse symbolique d'ordonnançabilité.

Ordonnancement abstrait

Un ordonnancement abstrait des acteurs est construit dans une première étape. Il consiste en un ensemble de contraintes d'ordonnancement non temporelles ; plus précisément, un ensemble de relations entre les horloges abstraites d'activation des tâches. La figure 2.(a) représente un graphe flot de données composé de deux acteurs, p_1 et p_2, et d'un buffer $e = (p_1, p_2, x, y)$ de taille $\delta(e)$ et qui contient initialement $\theta(e)$ jetons. Les fonctions $x, y : \mathbb{N}_{>0} \to \mathbb{N}$ indiquent les taux de production et de consommation du buffer e ; durant le j^e actionnement du producteur p_1, le nombre de jetons produits sur e est égal à $x(j)$. Les taux de production et de consommation sont constants dans le modèle SDF et périodiques dans le modèle CSDF. Dans cette thèse, on traite de types de taux plus expressifs comme par exemple les taux ultimement périodiques.

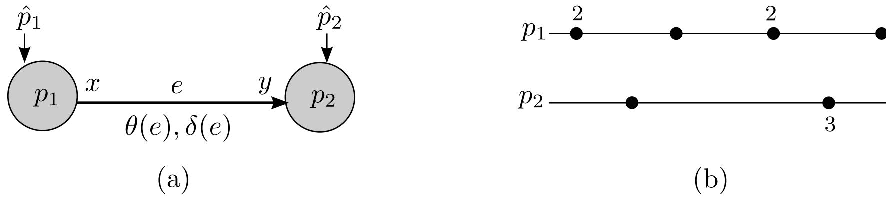


FIGURE 2 – (a) un graphe flot de données et (b) une relation d'activation.

Chaque acteur p_i est associé à une horloge d'activation $\hat{p_i}$ qui indique combien d'instances de p_i sont *activées* à un moment donné. Puisque cette première étape ne prend pas en compte le temps physique, nous sommes intéressés uniquement par l'ordre logique des activations d'où la notion de *relation d'activation*. La figure 2.(b) représente une relation d'activation entre les acteurs p_1 et p_2. La relation montre clairement l'ordre logique des activations ; par exemple, la première activation de p_2 est précédée par deux activations de p_1. Cependant, ça ne veut dire pas que la première instance de p_2 (dénotée par $p_2[1]$) ne commence son exécution que lorsque $p_1[2]$ s'acheva. Cela dépend de la stratégie d'ordonnancement et d'autres paramètres (par ex. priorités). Une relation d'activation peut être exprimée à l'aide de deux séquences infinies d'entiers. La construction d'un ordonnancement abstrait consiste à calculer toutes les relations d'activation entre les acteurs adjacents de façon à satisfaire les conditions suivantes.

Analyse des overflows : Un overflow se produit lorsqu'une tâche (i.e. un acteur) tente d'écrire des jetons sur un buffer plein. Pour éliminer ce type d'exception, il faut garantir que le nombre de jetons accumulés dans chaque buffer à chaque instant est inférieur ou égal à la taille du buffer. Si $\oplus x(j) = \sum_{i=1}^{j} x(i)$, alors l'analyse des overflows pour un buffer $e = (p_i, p_k, x, y)$ peut être décrite par l'équation suivante :

$$\forall j \in \mathbb{N}_{>0} : \theta(e) + \oplus x(j) - \oplus y(j') \leq \delta(e) \quad (1)$$

de telle sorte que $p_k[j']$ est la dernière instance de p_k qui certainement lit ses entrées à partir de e avant que $p_i[j]$ ne commence l'écriture de ses résultats sur le buffer. Le calcul de j' dépend de la relation d'activation entre p_i and p_k mais aussi de la stratégie d'ordonnancement. Ce calcul ne doit pas prendre en compte le temps physique (par ex. le temps d'exécution pire-cas des tâches) mais plutôt les scénarios pire-cas des overflows.

Analyse des underflows : Un underflow se produit lorsqu'une tâche tente de lire à partir d'un buffer vide. Il faut donc garantir que le nombre de jetons accumulés dans chaque buffer à chaque instant est supérieur ou égal à zéro. Formellement, il faut assurer que

$$\forall j \in \mathbb{N}_{>0} : \theta(e) + \oplus x(j') - \oplus y(j) \geq 0 \tag{2}$$

de telle sorte que $p_i[j']$ est la dernière instance de p_i qui écrit tous ses résultats sur e avant que $p_k[j]$ ne commence la lecture de ses entrées à partir du buffer.

Consistance : Si le graphe flot de données ne contient pas de cycles non orientés, alors chaque relation d'activation peut être calculée séparément des autres relations en utilisant les équations 1 et 2. Cependant, s'il y a des cycles non orientés dans le graphe, il faut assurer la cohérence des relations calculées afin d'éviter les problèmes de causalité. La figure 3 représente un ordonnancement abstrait inconsistant qui contient des problèmes de causalité. En effet, on peut déduire à partir de la relation d'activation entre p_1 et p_2 et de la relation entre p_2 et p_3 que la 4e activation de p_3 précède strictement la 2e activation de p_1. Par contre et partant de la relation entre p_3 et p_1, la 2e activation de p_1 précède strictement la 2e activation de p_3.

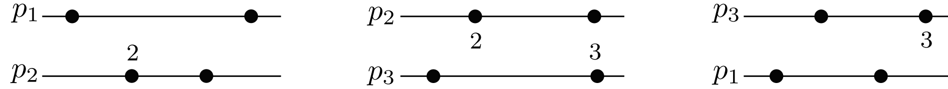


FIGURE 3 – Un ordonnancent abstrait inconsistant.

La première partie du chapitre 1 présente en détails l'approche d'ordonnancement abstrait pour des relations d'activation arbitraires. Néanmoins, nous sommes plutôt intéressés par des ordonnancements périodiques ; d'où l'introduction de la notion de relation d'activation *affine*. Une relation affine entre p_i et p_k est décrite par trois paramètres (n, φ, d) tels que toutes les d activations de p_i il y a n activations de p_k (i.e. n et d encodent la relation entre les vitesses des deux tâches) tandis que le paramètre φ encode la différence entre les phases de p_i et p_k. La figure 4 représente une relation d'activation affine de paramètres $(4, 2, 3)$.

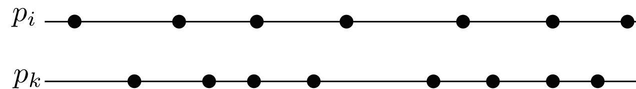


FIGURE 4 – Une relation d'activation affine de paramètres $(4, 2, 3)$.

Nous présentons dans la deuxième partie du chapitre 1 une méthode pour calculer les ordonnancements abstraits et affines des graphes dont les taux de production et de

consommation sont ultimement périodiques. Cette méthode se base sur la programmation linéaire en nombres entiers et a comme objectif la minimisation de la somme totale des tailles des buffers. Par ailleurs, deux stratégies d'ordonnancement sont considérées : EDF et ordonnancement à priorités fixes.

Analyse symbolique d'ordonnançabilité

Chaque acteur p_i est assigné à une tâche temps-réel périodique caractérisée par un temps d'exécution pire-cas C_i, une période π_i, une phase r_i, une échéance d_i, une priorité (dans le cas d'ordonnancement à priorités fixes), et le processeur sur lequel elle s'exécute (dans le cas d'ordonnancement multiprocesseur partitionné). L'ordonnancement abstrait décrit les relations entre les paramètres des tâches mais il ne calcule pas les valeurs de ces paramètres. L'analyse symbolique d'ordonnançabilité consiste à calculer ces paramètres de telle façon à :

- *Respecter l'ordonnancement abstrait :* Si la relation affine entre deux acteurs p_i et p_k est de paramètres (n, φ, d), alors les caractéristiques temporelles des deux tâches doivent satisfaire les deux contraintes suivantes qui assurent que les activations d'un acteur sont séparées par une période constante.

$$d\pi_i = n\pi_k \qquad\qquad r_k - r_i = \frac{\varphi}{n}\pi_i$$

- *Optimiser les performances :* Dans cette thèse, on considère deux métriques de mesure de performance des systèmes embarqués temps-réel : le débit (ou de manière équivalente, le facteur d'utilisation du processeur $U = \sum_{p_i} \frac{C_i}{\pi_i}$) et la somme totale des tailles des buffers. Malheureusement, la maximisation du débit est en conflit avec la minimisation des tailles des buffers. Les paramètres qui influencent principalement ces performances sont : les périodes, les priorités dans le cas d'ordonnancement à priorités fixes, les échéances dans le cas d'ordonnancement EDF, et l'allocation des processeurs dans le cas d'ordonnancement multiprocesseur partitionné. Nous proposons plusieurs techniques d'affectation des priorités et d'adaptation des échéances pour optimiser les performances.

- *Assurer l'ordonnançabilité :* On ne peut pas appliquer directement les tests standards d'ordonnançabilité car les paramètres temporels des tâches (principalement les périodes) sont inconnus. C'est pourquoi on a modifié ces tests afin de pouvoir calculer les paramètres temporels des tâches pour lesquelles le système est ordonnançable sur une architecture et pour une stratégie d'ordonnancement données. Nous proposons plusieurs algorithmes ; à titre d'exemple, `SQPA` pour l'ordonnancement EDF monoprocesseur, `SRTA` pour l'ordonnancement monoprocesseur à priorités fixes, `SQPA-FFDBF` pour l'ordonnancement EDF multiprocesseur global, etc.

Pour conclure, nous montrons l'efficacité de notre approche en utilisant des graphes issus de cas réels, ainsi que des graphes générés aléatoirement. Nous proposons aussi un flot de conception des applications Java pour les systèmes critiques (SCJ), basé sur l'approche décrite précédemment.

Chapter 1

Design and Verification

Contents

1.1 Generalities . . . 11

1.2 Dataflow models of computation . . . 13

1.2.1 Kahn process networks . . . 14

1.2.2 Bounded execution of KPNs . . . 15

1.2.3 Dataflow process networks . . . 17

1.2.4 Specific dataflow graph models . . . 18

1.2.5 Dataflow synchronous model . . . 21

1.3 Static analysis of (C|H)SDF graphs . . . 26

1.3.1 Reachability analysis . . . 26

1.3.2 Timing analysis . . . 29

1.3.3 Memory analysis . . . 33

1.4 Real-time scheduling . . . 35

1.4.1 System models and terminology . . . 36

1.4.2 EDF schedulability analysis . . . 39

1.4.3 Fixed-priority schedulability analysis . . . 43

1.4.4 Symbolic schedulability analysis . . . 45

1.4.5 Real-time scheduling of dataflow graphs . . . 47

1.5 Real-time calculus . . . 48

1.6 Conclusion . . . 49

1.1 Generalities

This introductory section presents general notions about some types of computer-based systems starting from the most general to the most specific.

Embedded systems

Embedded systems are ubiquitous in our daily lives. They are present in many industries, including transportation, telecommunication, defense, and aerospace. They have changed the way we communicate, the way we conduct business - in short, our inventions are changing who we are.

What are embedded systems? There are several definitions in the literature [97, 125, 78] that may agree with our vision. We believe that an embedded system is one that integrates software with hardware to accomplish a dedicated function that is subject to physical constraints. Such constraints are arising from either the behavioral requirements (e.g. throughput, deadlines, etc.) or the implementation requirements (e.g. memory, power, processor speed) of the system.

Separate design of software and hardware does not work for embedded systems where techniques from both fields should be combined into a new approach that should not only meet physical constraints but also reduce the development cost and the time to market.

Reactive systems

A reactive system is a system that maintains a permanent interaction with its environment [94]. Unlike interactive systems (e.g. database management systems) which interact at their own speed with users or with other programs, reactive systems are subject to some reaction constraints represented in the way the external environment dictates the rhythm at which they must react. Reactive systems may enjoy functional determinism. This key property means that for a given sequence of inputs, a system will always produce the same sequence of outputs.

All reactive systems are embedded systems; however, not all embedded systems are reactive systems. For example, proactive systems (e.g. autonomous intelligent cruise control) extend reactive systems with more autonomy and advanced capabilities - they anticipate and take care of dynamically occurring (not necessarily a priori predicted) situations.

Depending on how reactive systems acquire inputs, they can be either event-driven or sampled systems. An event-driven system waits for a stimulus to react. So, if its response time is too long, it may miss some subsequent events. On the other hand, a sampled system requires its inputs at equidistant time instants according to some physical requirements. Examples of sampled systems are flight control systems and real-time signal processing systems. In this thesis, we are mainly interested in sampled systems.

Real-time systems

Real-time systems are reactive systems for which the correct behavior depends not only on the outputs but also on the time at which the results are produced. A real-time task is usually characterized by a deadline before which the task must complete its execution. It is said to be *hard* if missing the deadline may cause catastrophic consequences; while

it is said to be *soft* if missing the deadline is not catastrophic but the the result becomes less valuable - e.g. it decreases the quality of a video game. Finally, the task is said to be *firm* if a late response is worthless.

Real-time computing is not just *fast* computing or matter of good average case performance, since these cannot guarantee that the system will always meet its deadlines in the worst-case scenario. Real-time computing should rather be a predictable computing [51]. The need for predictability should not be confused with the need for *temporal determinism* where the response times of the system can be determined for each possible state of the system and set of the inputs.

Safety-critical systems

A safety-critical system is a system whose failure may cause serious injury to people, loss of human life, significant property damage, or large environmental harm. Flight control systems, railway signaling systems, robotic surgery machines, or nuclear reactor control systems naturally come to mind. But, something simple as traffic lights can also be considered as safety-critical since giving green lights to both directions at a cross road could cause human death.

Many disciplines are involved in safety-critical systems design [111]: domain engineering, embedded systems engineering, safety engineering, reliability engineering, etc. Unlike reliability engineering which is concerned with failures and failure rate reduction, the primary concern of safety engineering is the management of hazards, i.e. identifying, analyzing, and controlling hazards throughout the life cycle of a system in order to prevent or reduce accidents. In a domain such as avionics, a system is designed to have at most 10^{-9} accidents per hour.

All safety-critical systems are real-time systems because they must respond, as required by the safety analysis, to a fault by the fault tolerance time which is the length of time after which the fault leads to an accident [72].

1.2 Dataflow models of computation

It is easier to explain the concept of "model of computation" by giving some examples rather than to search for a precise essential definition. State machines, timed Petri nets, synchronous models, Kahn process networks, and communicating sequential processes are typical examples of models of computation. A model of computation consists of a set of laws that govern the interaction of components in a design. It usually defines the following elements [113]- ontology: what is a component? epistemology: what knowledge do components share? protocols: how do components communicate? lexicon: what do components communicate?

A model of computation for embedded real-time systems should deal with concurrency and time. A classification of models of computation according to their timing abstraction can be found in [99]. They can be continuous-time, discrete-time, synchronous, or untimed models.

Dataflow models of computation are characterized by a data-driven style of control; data are processed while flowing through a network of computation nodes. Therefore, communication and parallelism are very exposed. Dataflow programming languages can be traced back to the 70's with a great advancement in the underlying models of computation and visual editors since then [102].

In this chapter, we will present the three major variants of dataflow models in the literature, namely, Kahn process networks [105], Dennis dataflow [70], and synchronous languages [21, 94]. Mathematical notations and definitions used in this chapter can be found in Appendix A.

1.2.1 Kahn process networks

A Kahn process network (KPN) is a collection of concurrent processes that communicate only through unidirectional first-in, first-out (FIFO) channels. A process is a deterministic sequential program that executes, possibly forever, in its own thread of control. It cannot test for the presence or absence of data (also called tokens) on a given input channel. Therefore, a process will block if it attempts to read from an empty channel. Communications in KPNs are asynchronous; i.e. the sender of the token needs not to wait for the receiver to be ready to receive it. Furthermore, writing to a channel is a non-blocking operation since channels are assumed to be unbounded.

KPNs are deterministic; i.e. the history of tokens produced on channels does not depend on either the execution order or the communication latencies. Gilles Kahn had provided an elegant mathematical proof of determinism using a denotational framework [105].

A Denotational semantics

A denotational semantics explains *what* input/output relation a KPN computes. A Kahn process is a *continuous* functional mapping from input streams into output streams. A stream or a *history* of a channel is the sequence of tokens communicated along that channel. It can be the input (and the output) of at most one process. The basic patterns of deterministic composition of processes are described in [115].

Every continuous process is *monotone* but not vice versa. Monotonicity of a process means that the process needs not to wait for all its inputs to start computing outputs, but it can do that iteratively. Thus, monotonicity implies a notion of *causality* since future inputs concern only future outputs. Furthermore, continuity prevents a process from waiting forever before sending some outputs.

Let $\{s_1, \ldots, s_N\}$ be the set of streams of the network; and let $\{f_1, \ldots, f_N\}$ be the set of terms built out of processes such that for each stream s_i, we have that $s_i = f_i(s_1, \ldots, s_N)$. If we take S to be equal to the N-tuple $[s_1, \ldots, s_N] \in \mathcal{A}^N$, then the network can simply be described as $S = F(S)$ with $F : \mathcal{A}^N \longrightarrow \mathcal{A}^N$. It is worth mentioning that feedback loops, which can be used to model local states, fit naturally in this description framework.

The denotational semantics of the KPN is the solution of the equation system $S =$

$F(S)$. Therefore, it is only a matter of computing a least fixed point solution. Since Kahn processes are continuous over *cpos*, there is a ***unique*** least fixed point solution; i.e. the network is deterministic. According to the least fixed point theorem for monotone functions on *cpos*, the minimum solution of the system can be iteratively computed by $S^{j+1} = F(S^j)$ till stabilization such that S^0 consists of the initial streams of the network.

B Operational semantics

Despite its mathematical beauty, the denotational semantics is not suitable for reasoning about implementation related aspects such as buffer sizes and potential deadlocks. In the past decades, several operational semantics of KPNs were given, for instance, in the form of labeled transition systems [81], I/O automata [133], or concurrent transition systems [177]. Like it had been proved, the behavior of a KPN according to these operational semantics corresponds to the least fixed-point solution given by Kahn. This equivalence is referred to as the *Kahn principle*.

C Limitations of determinism

A Kahn process network models a functional deterministic system; however, it is widely accepted that there are many systems that require non-determinism such as resource management systems. KPNs are unsuitable for handling asynchronous events or dealing with timing properties. To model such behaviors, one has to extend the model with additional features that may break the property of determinism.

Non-determinism can be added to KPNs by any of the following methods described in [118]: (1) allowing a process to test inputs for emptiness; a solution that was adopted in [69, 136], (2) allowing multi-writer channels and/or multi-reader channels, (3) allowing shared variables; a solution that was used in [107], and (4) allowing a process to be internally non-deterministic.

The non-deterministic merge is essential for modeling reactive systems. Its behavior consists in moving tokens whenever they arrive on any of its two inputs to its unique output. Hence, the output sequence depends on the arrival times of the input tokens. This merge process is not a Kahn process since it has to be either non-monotone or unfair [47].

1.2.2 Bounded execution of KPNs

Kahn process networks are deterministic; i.e. the order in which the processes execute, assuming a *fair* scheduler, does not affect the final result. Fairness states that in an infinite execution, a ready process must not be postponed forever. Even if KPNs only allow modeling of functional deterministic systems, they are a very expressive model. In fact, they are *Turing-complete*.

Due to Turing-completeness, some interesting properties of KPNs are undecidable [48]. For instance, the boundedness of an arbitrary KPN, i.e. whether it can be executed with bounded internal channels or not, is undecidable. Another undecidable problem is

deadlock-freedom which states that, whatever the input sequences, the KPN will never deadlock. Indeed, the halting problem of Turing machines, known to be undecidable, can be reduced to these problems.

The KPN model is not amenable to compile-time scheduling since it is not possible, in the general case, to decide boundedness in finite time. Thus, it is necessary to resort to the dynamic scheduling which has infinite time to find a bounded memory solution, if any. The behavior of a dynamically scheduled network must conform to its denotational semantics. Authors of [150, 16, 81] have defined some requirements, described below, for correct schedulers.

A Correctness criteria

A scheduler is correct with respect to Kahn semantics if and only if every execution the scheduler may produce is sound, complete, and bounded whenever it is possible.

Soundness An execution is sound if and only if the produced tokens are not different from the formal ones. So, if s is the actual stream produced on a given channel and $s^{\#}$ is the corresponding formal stream predicted by the denotational semantics, then $s \sqsubseteq s^{\#}$.

Completeness Let $s_0 \sqsubseteq s_1 \sqsubseteq \cdots$ be the partial streams produced by the execution on a given channel such that any progress is done in finite time. The execution is complete if and only if $s^{\#} = \bigsqcup\{s_0, s_1, \ldots\}$.

Boundedness An execution is bounded if and only if the number of accumulated tokens on each internal channel does not exceed a bound. Following [150], a KPN is *strictly* bounded if every complete execution of the network is bounded; it is bounded if there is at least one bounded execution; and it is unbounded if any execution requires unbounded memory.

B Boundedness and artificial deadlocks

The assumption about unbounded channels is clearly unrealistic. Real implementations bound channel capacities and impose blocking write operations; i.e. a process may block if it attempts to write on a full channel. This limitation of the model does not impact its determinism. In fact, any arbitrary KPN can be easily transformed into a strictly bounded network by adding a feedback channel for each internal channel. Before writing a token to an internal channel, a process must first read one token from the corresponding feedback channel. Dually, after reading a token from a channel, it has to write one token on the feedback channel. This way, the size of the channel is bounded by the number of initial tokens in the feedback channel.

This new operational semantics could introduce artificial deadlocks. This is the case when a subset of processes are blocked in a deadlock cycle, with at least one process being blocked on writing to a full channel. Since it is impossible to compute channel capacities at compile-time, one solution is to dynamically increase capacities when artificial deadlocks occur. Thus, dynamic scheduling requires run-time detection and

resolution of artificial deadlocks. It has been proved in [81] that an artificial deadlock cannot be avoided by only changing the scheduling and without increasing some channel capacities.

The dynamic scheduling strategy proposed by Parks [150] executes the KPN with initially small channel capacities. If an *artificial* global deadlock occurs, then the scheduler increases the capacity of the smallest channel and continues. Hence, if the KPN is bounded, then its execution will require a bounded memory. Geilen and Basten [81] have noticed that the strategy of Parks is not complete because it deals only with global deadlocks. Indeed, it gives priority to non-terminating execution over bounded execution, and to bounded execution over complete execution. They proposed therefore a deadlock resolution algorithm, built upon the notion of chains of dependencies, that also handles artificial local deadlocks. This scheduling strategy is correct for *effective* KPNs where a produced token on an internal channel is ultimately consumed.

C Dynamic scheduling policies

Dynamic scheduling policies can be classified as data-driven policies (i.e. eager execution), or demand-driven policies (i.e. lazy execution), or a combination of both.

Demand-driven scheduling The activation of a process is delayed until its output is needed by another process. Kahn and MacQueen described a demand-driven scheduling technique in [106] where the process responsible of sending the results to the environment is selected to drive the whole network. Upon an attempt to read from an empty channel, the consumer blocks and the producer process for that channel is activated. This latter will be suspended after producing the necessary data, and then the waiting consumer process will be again activated. In a technique called *anticipation*, the producer process may continue producing data on the channel in parallel with the consumer until it reaches the threshold of the channel called the *anticipation coefficient*.

Data-driven scheduling A process is activated as soon as sufficient data is available on its input ports. This policy satisfies the completeness criterion, since processes only block on reading from empty channels. The data-driven policy may lead to unbounded accumulation of tokens on internal channels. Pingali and Arvind proved that it is possible to transform a network so that a data-driven execution of the resulted network is equivalent to a demand-driven execution of the original one [153, 154]. This implies that a demand-driven scheduling policy may or may not satisfy the boundedness criterion.

1.2.3 Dataflow process networks

A dataflow process network (DPN) is a graph where nodes are dataflow actors and edges are FIFO channels. An actor, when it fires, consumes finite number of input tokens and produces finite number of output tokens. A set of firing rules indicates when the actor is enabled to fire. This style of dataflow models, introduced by Dennis [70], has influenced concurrent programming languages and computer architectures for several years.

As shown later, repeated firings of an actor form a particular type of Kahn process called dataflow process. It is very convenient to break down a process to a sequence of

firings in order to enable efficient implementations and formal analyses. However, the relation between the Kahn's denotational semantics and the operational data-driven semantics of DPNs became clear only after the outstanding work of Lee and Parks [115, 150, 118].

Mathematically speaking, a dataflow actor with m input and n output channels consists of a *firing* function $f : \mathcal{A}^m \longrightarrow \mathcal{A}^n$ and a finite set R of firing rules. A firing rule is just an m-tuple that is "satisfied" if each sequence of the tuple is a prefix of the sequence of tokens accumulated in the corresponding input channel. In other words, it indicates what tokens must be available at the inputs for the actor to be enabled. The code of the firing function will be executed each time a rule is satisfied, and tokens will be consumed as specified by that rule. Clearly at most one firing rule should be satisfied at each time in order to have a deterministic behavior of the actor. Therefore, for all $r, r' \in R$, if $r \neq r'$, then $r \sqcup r'$ must not exist.

A set of firing rules is said to be *sequential* if it can be implemented as a sequence of blocking read operations. Lee and Parks proposed in [118] a simple algorithm to decide whether a set of firing rules is sequential or not. Sequential firing rules are a sufficient condition for a dataflow process to be continuous. Despite the fact that not every set of firing rules which satisfies the above mentioned condition about upper bounds is sequential, that condition is more restrictive than what is really necessary for a dataflow to be continuous. A weaker condition (commutative firing rules) is given in [115] which states that if the upper bound $r \sqcup r'$ exists, then the order in which these two rules are used does not matter; i.e. $f(r).f(r') = f(r').f(r)$ and $r \sqcap r' = \epsilon_m$. Unlike sequential firing rules, commutative rules enable compositionalilty; in the sense that the composition of two actors with commutative firing rules is an actor with commutative firing rules.

Relation to KPNs

Each actor, with a function f and a set of firing rules R, is associated with a functional $F : [\mathcal{A}^m \longrightarrow \mathcal{A}^n] \longrightarrow [\mathcal{A}^m \longrightarrow \mathcal{A}^n]$ such that

$$\forall g \in [\mathcal{A}^m \longrightarrow \mathcal{A}^n] : \forall S \in \mathcal{A}^m : F(g)(S) = \begin{cases} f(r).g(S') & \textit{if}\ \exists r \in R : S = r.S' \\ \epsilon_m & \textit{otherwise} \end{cases}$$

As proved in [115], F is a continuous and closed function on the CPO $([\mathcal{A}^m \longrightarrow \mathcal{A}^n], \sqsubseteq)$ and has therefore a least fixed-point that can be computed iteratively by $\forall S \in \mathcal{A}^m : g_0(S) = \epsilon_n$ (g_0 is trivially continuous) and $g_{n+1} = F(g_n)$. Clearly, if $S = r_1.r_2.\cdots$, then $g_0(S) = \epsilon_n, g_1(S) = f(r_1), g_2(S) = f(r_1).f(r_2), \ldots$ In words, the dataflow process (i.e. the least fixed-point) is constructed by repeatedly firing the actor. This least fixed-point is continuous and it is therefore a Kahn process.

1.2.4 Specific dataflow graph models

Due to Turing-completeness of dataflow graphs, their boundedness and static scheduling are generally undecidable. A bunch of restricted models that trade off expressiveness

for decidability have been developed over the past years. They can be classified into two main categories: static dataflow graphs [93] and dynamic dataflow graphs [29].

A Static dataflow graphs

Static dataflow models restrict actors so that on each port and at each firing, they produce and consume a compile-time known number of tokens. These models are amenable for construction of static schedules (with finite descriptions) that execute the graphs in bounded memory.

The synchronous dataflow (SDF) model [117] is widely used for embedded systems design, especially for digital signal processing systems. In SDF, the production (or consumption) rate of each port is constant; i.e. an actor produces (or consumes) a fixed number of tokens on each port. If all the rates in the graph are equal to 1, then the graph is said to be a homogeneous synchronous dataflow (HSDF) graph. HSDF graphs are equivalent to marked graphs [140] while SDF graphs are equivalent to weighted marked graphs [187]. Both marked graphs and weighted marked graphs are subclasses of Petri nets, whose literature provides many useful theoretical results. Computation graphs [109] are similar to SDF graphs; however, each channel is associated with a threshold that can be greater than its consumption rate. An actor can fire only if the number of accumulated tokens on each input channel exceeds its threshold. Cyclo-static dataflow (CSDF) model [41] is a generalization of SDF where the number of produced (or consumed) tokens on a port may vary from one firing to another in a cyclic manner.

The SDF model assumes that both the producer and the consumer of a channel manipulate the same data type. However, in multimedia applications, it is natural to use composite data types such as video frames so that, for example, a node may consume or produce only a fraction of the frame each time. In the fractional rate dataflow (FRDF) model [146], an actor can produce (or consume) a fractional number of tokens at each firing which leads generally to better buffering requirements compared to the equivalent SDF graph. The FRDF model gives a statistical interpretation of a fraction $\frac{x}{y}$ associated to a given port in case of atomic data types; i.e. the actor produces (or consumes) x tokens each y firings but without assuming any knowledge on the number of tokens produced (or consumed) each time. Therefore, the scheduling algorithm should consider the worst-case scenarios.

More varieties of the SDF model, that tailor the model for specific needs, can be found in the literature. In the scalable synchronous dataflow model [160], the number of produced (or consumed) tokens on a given port can be any integer multiple of the predefined fixed rate of that port. So, all rates of an actor will be scaled by a scale factor. The static analysis must determine the scale factor of each actor in a way that compromises between function call overheads and buffer sizes.

B Dynamic dataflow graphs

Some modern applications have production and consumption rates that can vary at run time in ways that are not statically predictable. Dynamic models provide the required

expressive power, but at cost of giving up powerful static optimization techniques or guarantees on compile-time bounded scheduling.

Boolean dataflow (BDF) model [48] is an extension of SDF that allows data-dependent production and consumption rates by adding two control actors called select and switch. The switch actor consumes one boolean token from its control input and one token from the data input, and then copies the data token to the first or second output port according to the value of the control token. The behavior of the select actor is dual to that of the switch actor; both of them combined may allow us to build if-then-else constructs and do-while loops. BDF model is Turing-complete [48]; hence, the questions of boundedness and liveness are undecidable. Nevertheless, the static analysis is yet possible for many practical problems. A variant of the BDF model is the integer-controlled dataflow (IDF) model [49] in which we can model an actor that consumes an integer token and then produces a number of tokens equal to that integer.

Scenario aware dataflow (SADF) model [188, 181] extends the SDF model with scenarios that capture different modes of operation and express hence the dynamism of modern streaming applications. Production and consumption rates and execution times of actors may vary from one scenario to another. The SADF model distinguishes data and control explicitly by using two kinds of actors: kernels for the data processing parts and detectors to handle dynamic transitions between scenarios. Furthermore, there are two kinds of channels: control channels which carry scenario-valued tokens and usual data channels. Detectors contain discrete-time Markov chains to capture occurrences of scenarios and allow hence for average-case performance analysis.

Parameterized synchronous dataflow (PSDF) [28] and schedulable parametric dataflow (SPDF) [76] models are examples of parameterized dataflow which is a meta-modeling approach for integrating dynamic parameters (such as parametric rates) in models that have a well-defined concept of iteration (e.g. SDF and CSDF). SPDF extends SDF with parametric rates that may change dynamically and it is therefore necessary to statically check consistency and liveness for all possible values of the parameters. In addition to the classical analyses of SDF, we have also to check whether dynamic update of parameters is safe in terms of boundedness. This analysis is possible since parameters can be updated only at the boundaries of (local) iterations.

C Comparison of models

Dataflow models can be compared with each other using three features [179]: expressiveness and succinctness, analyzability and implementation efficiency. These features can be illustrated by the following examples. BDF model is more expressive than the CSDF model since this latter does not allow modeling of data-dependent dynamic production and consumption rates. The CSDF, SDF, HSDF models have the same expressiveness; any behavior that can be modeled with one of them can be modeled with the two other models. Nevertheless, the resulted graphs have different sizes with CSDF graphs being the most succinct. Transformation of (C)SDF graphs into equivalent HSDF graphs use an unfolding process that replicates each actor possibly an exponential number of times [175, 116, 41]. A transformation of CSDF graphs into SDF graphs such that

each CSDF actor is mapped to a single SDF actor was presented in [152]; however, this transformation may create deadlocks, covers unnecessary computations, and exposes less parallelism.

HSDF model is more analyzable than the (C)SDF model in sense that existing analysis algorithms for HSDF graphs have lower complexities than the corresponding algorithms for (C)SDF graphs with the same number of nodes. Implementation efficiency concerns the complexity of the scheduling problem and the code size of the resulting schedules. For example, scheduling of computation graphs is more complex than scheduling of SDF graphs because of the threshold constraint on consumption.

1.2.5 Dataflow synchronous model

KPNs and Dataflow process networks are closely related, while dataflow synchronous models are quite different. There are a bunch of synchronous languages (e.g. SIGNAL [92], ESTEREL [24], LUSTRE [95], LUCID SYNCHRONE [54]) dedicated to reactive systems design that rely on the same assumptions (*the synchronous paradigm*) [19, 20]:

1. Programs progress via an infinite sequence of reactions. The system is viewed through the chronology and simultaneity of the observed events, which constitute a logical time, rather than the chronometric view of the execution progress. The synchronous hypothesis therefore assumes that computations and communications are instantaneous. This hypothesis is satisfied as long as a reaction to an input event is fast enough to produce the output event before the acquisition of the next input events. The synchrony hypothesis simplifies the system design; however it implies that only bounded KPNs can be specified (by programming processes that act as bounded buffers).

2. Testing inputs for emptiness is allowed. Hence, decisions can be taken on the basis of the absence of some events. Synchronous programs can have therefore a non-deterministic behavior.

3. Parallel composition is given by taking the pairwise conjunction of associated reactions, whenever they are composable.

A Abstract clocks

The denotational semantics of both untimed models of computation (e.g. KPNs) and timed models (e.g. dataflow synchronous model) can be expressed in a single denotational framework ("meta-model"): the tagged signal model [119]. The main objectives of this model is to allow comparison of certain properties of several models of computation, and also to homogenize the terms used in different communities to mean sometimes significantly different things (e.g. "signal", "synchronous"). The basic idea of this framework is to couple streams (or sequences) with a tag system and consider hence signals instead of untimed streams. The tag system, when specifying a system, should not mark physical time but should instead reflect ordering introduced by causality. The tag system presented in this section is tailored to the dataflow synchronous model.

Definition 1.1 (Tag system)**.** A tag system is a complete partial order $(\mathcal{T}, \sqsubseteq)$. It provides a discrete time dimension that corresponds to logical instants according to

which the presence and absence of events can be observed.

Definition 1.2 (Events, Signals)**.** An event is a pair $(t, a) \in \mathcal{T} \times A$ such that A is a value domain. A signal s is a partial function $s : C \rightharpoonup A$ with C a chain in $\mathcal{T}$. It associates values with totally ordered observation points.

Definition 1.3 (Abstract clock)**.** The abstract clock of a signal $s : C \rightharpoonup A$ is its domain of definition $\hat{s} = \mathrm{dom}(s) \subseteq C$.

Usual set operations and comparisons can be naturally applied on clocks. Relations between clocks can be deduced from the different operations on signals (e.g. merge, select). The relational synchronous language SIGNAL allows explicit manipulation of clocks. Table 1.1 shows the essential SIGNAL operators and what relations should relate clocks. Both the select and merge operators are not Kahn processes. Unlike LUSTRE and ESTEREL, SIGNAL is a multi-clocked model; i.e. a master clock is not needed in the system. Therefore, two signals may have totally unrelated abstract clocks; such as the operands of the merge process.

Construct	clock relations	semantics
Stepwise extensions $r = f(s_1, \ldots, s_n)$	$\hat{r} = \hat{s}_1 = \cdots = \hat{s}_n$	$\forall t \in \hat{r} : r(t) = f(s_1(t), \ldots, s_n(t))$
Delay $r = s\$\,\mathrm{init}\, a_0$	$\hat{r} = \hat{s}$	$\forall t \in \hat{r} : r(t) = \begin{cases} a_0 & \textit{if } t = \bigsqcap \hat{r} \\ s(t^-) & \textit{otherwise} \end{cases}$ t^- and t are successive tags in $\hat{r}$
Select $r = s \,\mathrm{when}\, b$	$\hat{r} = \hat{s} \cap \{t \in \hat{b} \mid b(t) = \mathit{true}\}$	$\forall t \in \hat{r} : r(t) = s(t)$
Merge $r = s_1 \,\mathrm{default}\, s_2$	$\hat{r} = \hat{s}_1 \cup \hat{s}_2$	$\forall t \in \hat{r} : r(t) = \begin{cases} s_1(t) & \textit{if } t \in \hat{s}_1 \\ s_2(t) & \textit{otherwise} \end{cases}$

Table 1.1: SIGNAL elementary processes

Elementary relations can be combined to produce more sophisticated clock transformations. One important clock transformation is the affine transformation, described in [168], and used extensively in this thesis. A subclass of affine relations between abstract clocks was used in [79] to address time requirements of streaming applications on multiprocessor systems on chip.

Definition 1.4 (Affine transformation)**.** An affine transformation of parameters (n, φ, d) applied to the clock $\hat{s}$ produces a clock $\hat{r}$ by inserting $(n-1)$ instants between any two successive instants of $\hat{s}$, and then counting on this fictional set each dth instant, starting with the $(\varphi + 1)^{th}$ instant.

B Quasi-synchrony and N-synchronous

Two signals s_1 and s_2 are said to be *synchronous* if and only if $\hat{s}_1 = \hat{s}_2$. Synchronous dataflow languages provide a type system, called *clock calculus*, that ensures that a signal

can be assigned only to another signal with the same clock (i.e. buffer-less communication). This does not however mean that a signal cannot be delayed. Mitigation of this strong requirement has been the subject of some works motivated by the current practice in real-time systems development; especially real-time systems that consist of a set of *periodic* threads which communicate asynchronously.

Quasi-synchrony is a composition mechanism of periodic threads that tolerates small drifts between thread's release (i.e. activation) clocks [53]. If $\hat{s}$ and $\hat{r}$ are the activation clocks of two periodic threads, then their quasi-synchronous composition is such that between two occurrences of $\hat{s}$ there are at most two occurrences of $\hat{r}$, and conversely, between two occurrences of $\hat{r}$ there are at most two occurrences of $\hat{s}$. This definition was refined in [96] to enable synchronous occurrences. If t_1 and t_2 are two successive tags in $\hat{s}$, then $|\{t \in \hat{r} | t_1 \sqsubset t \sqsubseteq t_2\}| \leq 2$; Conversely, if t_1 and t_2 are two successive tags in $\hat{r}$, then $|\{t \in \hat{s} | t_1 \sqsubset t \sqsubseteq t_2\}| \leq 2$. A stream between two quasi-synchronously composed threads can be no more considered neither as an instantaneous (buffer-less) communication nor as a bounded FIFO; but rather as a shared variable. Hence, some values are lost if the sender is faster than the receiver and some values are duplicated if the receiver is faster than the sender. This non-flow-preserving semantics was taken further in the Prelude compiler [149] as shown later.

In the N-synchronous paradigm [62], two signals can be assigned to each other as long as a bounded communication is possible. Hence, the new synchronizability relation states that two clocks $\hat{s}$ and $\hat{r}$ are synchronizable if and only if for all $n \in \mathbb{N}_{>0}$, the nth tags of $\hat{s}$ and $\hat{r}$ are at bounded distance. Formally, $\forall t' \in \hat{s} : |\{t \in \hat{s} | t \sqsubseteq t'\}| - |\{t \in \hat{r} | t \sqsubseteq t'\}|$ is bounded. This model allows to specify any bounded KPN. Unlike the quasi-synchronous approach, communications in this model are flow-preserving.

C From synchrony to asynchrony

Although the synchronous hypothesis states that reactions are instantaneous, causality dependencies dictate in which order signals are evaluated within a reaction. For instance, the presence of a signal must be decided before reading its value. A schizophrenic cycle is one in which the presence of a signal depends on its value. More dependencies can be deduced from elementary processes. For example, computing values of $s_1, \ldots, s_n$ must precede computing value of $r = f(s_1, \ldots, s_n)$. Synchronous languages provide the necessary tools to check deadlock freedom; that is, there is no cycle in the dependencies graph.

Desynchronizing a signal to obtain an asynchronous communication consists in getting rid of the tags information and hence preserving only the stream. A synchronous process may take some decisions based on the absence of a signal; but it can no more test the emptiness of an input in an asynchronous implementation. The question that raises naturally is whether it is possible to resynchronize a desynchronized signal. An *endochronous* process [19] is a process that can incrementally infer the status (presence/absence) of all signals from already known and present signals. Therefore, an endochronous process should have a unique master clock so that all the other clocks are subsets of it. Endochrony is undecidable in the general case [19]. An endochronous pro-

cess is equivalent to a dataflow process with sequential set of firing rules; and it is hence a Kahn process. Weak endochrony [157] is a less restrictive condition than endochrony. A weakly endochronous process is equivalent to a dataflow process with commutative firing rules. Similarly to composition properties of dataflow actors, the composition of two endochronous processes is not in general an endochronous process; while the composition of two weakly endochronous processes results in a weakly endochronous process.

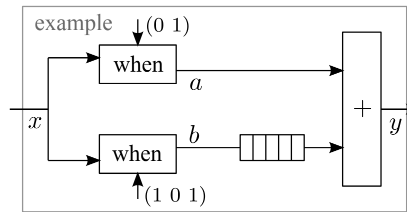
The compositional property of isochrony ensures that the synchronous composition of (endochronous) processes is equivalent to their asynchronous composition. It is an important property for designing globally asynchronous locally synchronous (GALS) architectures. Roughly speaking, a pair of processes is isochronous if every pair of their reactions which agree on present common signals also agree on all common signals.

D Lucy-n

Lucy-n [134] is an extension of the synchronous language LUSTRE with an explicit *buffer* construct for programming networks of processes communicating through bounded buffers. The synchronizability relation is the one defined by the N-synchronous paradigm. Lucy-n provides a clock calculus (defined as an inference type system) which can compute the necessary buffer sizes using a linear approximation of boolean clocks. We illustrate the semantics of Lucy-n using a simple example (Listing 1.1). The clock of each signal is denoted by a binary sequence such that 1 represents presence and 0 represents absence. The clock calculus handles only ultimately periodic clocks which are denoted syntactically by $u(v)$. The N-synchronous execution is presented in Table 1.2.

Listing 1.1: Example in Lucy-n.

```
let node example x=y where
  rec a=x when (0 1)
  and b=x when (1 0 1)
  and y= a+ buffer(b)
```



Since the $(+)$ operator is a stepwise operator, signals a and *buffer*(b) must have the same clock 1^ω *on* $(0\ 1)^\omega$. However, this program must be rejected because signal a and b are not synchronizable (as defined by the N-synchronous paradigm). Indeed, the difference between the nth tags of $\hat{a}$ and $\hat{b}$ are not at a bounded distance when n tends to infinity. So, the inserted buffer is unbounded.

Table 1.2: N-synchronous execution of Program 1.1.

signal	flow							clock
x	2	5	1	0	3	2	1	1^ω
$a = x$ when $(0\ 1)$		5		0		2		1^ω *on* $(0\ 1)^\omega$
$b = x$ when $(1\ 0\ 1)$	2		1	0		2	1	1^ω *on* $(1\ 0\ 1)^\omega$
buffer(b)		2		1		0		1^ω *on* $(0\ 1)^\omega$
y		7		1		2		1^ω *on* $(0\ 1)^\omega$

E Prelude

Prelude [149] is a real-time programming language build upon the synchronous language LUSTRE. Listing 1.2 represents a simple example; while Table 1.3 shows its semantics. The difference between Prelude and the synchronous languages consists mainly in the two following points.

Listing 1.2: Example in Prelude.

```
node foo (x: rate(10,0) ) returns (y)
  var a,b;
let
  (y,a) =Add (x,(1 fby b)*^2);
  b = INC(a/^2);
tel
```

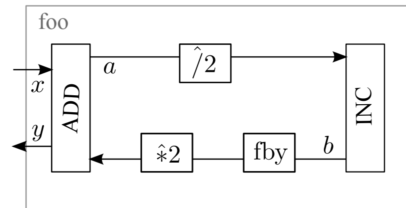


Table 1.3: Semantics of Program 1.2.

date	0	10	20	30	40	50	60
x	3	2	4	1	0	6	2
a	4	3	9	6	10	16	13
$a\hat{/}2$	4		9		10		13
b	5		10		11		14
1 fby b	1		5		10		11
(1 fby b)$\hat{*}$2	1	1	5	5	10	10	11

Real-time constraints: Real-time constraints represent environmental constraints and they are specified either on node inputs and outputs (e.g. periods, phases, deadlines) or on nodes (worst-case execution time). In Listing 1.2, signal x has a period of 10 and a first start time equals to zero. This implies that task ADD is a periodic task with a period equal to 10. The compiler checks the consistency of real-time constraints (e.g. inputs and outputs of a node have the same period) and earliest-deadline first schedulability of the specification.

Communication patterns: Rate transition operators (e.g. $\hat{*}$, $\hat{/}$) handle transition between nodes of different rates and hence enable the definition of user-provided communication patterns. For example, in Listing 1.2, flow a is under-sampled using operator $\hat{/}2$; which implies that node ADD is twice as fast as node INC. As in quasi-synchrony, communication is non-flow-preserving; i.e. when the producer is faster than the consumer, an adapter is added allowing some produced tokens to not be buffered; and when the consumer is faster, an adapter is added to allow for reading the same value several times. Furthermore, communications are deterministic (i.e. not affected by the earliest-deadline first scheduling policy and preemption). To realize this objective some deadlines are adjusted to enforce some precedences between jobs.

1.3 Static analysis of (C|H)SDF graphs

The HSDF, SDF, and CSDF models have been used as the underlying models of several Digital Signal Processing (DSP) programming environments, since they combine good levels of analyzability and expressivity. Indeed, properties such as boundedness and liveness are decidable. Furthermore, static-periodic scheduling of (C|H)SDF graphs has been the subject of an enormous number of works that have addressed the problem with respect to different non-functional constraints: throughput, buffering, latency, code size, etc. The proposed analyses abstract from the actual values of data that are being communicated, and focus only on the non-functional properties such as the distribution of tokens over channels. This section presents some existing analyses; and it is by no means a complete in-depth survey.

Definition 1.5 (Static dataflow graph)**.** A static dataflow graph is a directed graph $G = (P, E)$ consisting of a finite set of actors $P = \{p_1, \ldots, p_N\}$ and a finite set of one-to-one edges (or channels) E. A channel $e = (p_i, p_k, x, y) \in E$ connects the producer p_i to the consumer p_k such that the production (resp. consumption) rate is given by the infinite integer sequence $x \in \mathbb{N}^\omega$ (resp. $y \in \mathbb{N}^\omega$). For instance, the jth firing of actor p_i (denoted by $p_i[j]$) writes $x(j)$ tokens on channel e.

(H|C)SDF graphs are specific cases of static dataflow graphs. Each rate function x is a constant sequence (i.e. $x = a^\omega$ with $a \in \mathbb{N}_{>0}$) in the SDF model, a periodic sequence (i.e. $x = v^\omega$ with $v \in \mathbb{N}^*$ and $\|v\| > 0$) in the CSDF model, and equals to 1 (i.e. $x = 1^\omega$) in the HSDF model. Figure 1.1 shows a SDF graph, and its transformation into an equivalent HSDF graph, where a black dot on an edge represents an initial token in the channel.

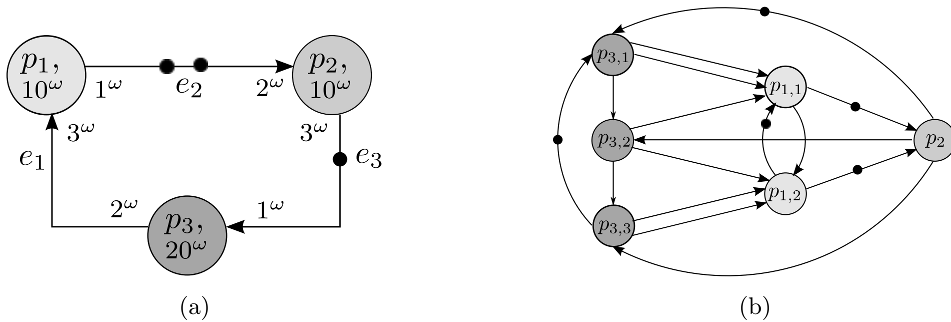


Figure 1.1: Example of (a) a cyclic SDF graph, and (b) its equivalent HSDF graph.

1.3.1 Reachability analysis

Since SDF graphs are equivalent to weighted marked graphs, many techniques from the Petri nets literature can be used to analyze them. A marking M (or channels state) is denoted by a vector whose i^{th} component $M(i)$ represents the number of tokens in

channel e_i. The initial marking M_0 contains for each channel e its number of initial tokens $\theta(e)$. Figure 1.2 represents the reachability graph of the SDF example (Figure 1.1(a)) where states are markings and transitions are actor firings. The SDF example is *strictly* bounded since its reachability graph is finite. However, if we delete channel e_1, then the reachability graph is infinite since actor p_1 is always enabled. This implies that the new SDF graph is not strictly bounded; but it is bounded since there exists at least one complete execution with bounded channels.

A dataflow graph is live if it has executions in which all actors are fired infinitely. An immediate result of the functional determinism of CSDF graphs (i.e. the scheduling order does not affect the results) is that if one execution deadlocks then all executions deadlock [84]. The reachability graph in Figure 1.2 is deadlock-free (i.e. $\forall M : \exists p_i \in P : p_i$ is firable at M) and live (i.e. $\forall p_i : \forall M : \exists M'$: the firing sequence that transforms M into M' contains p_i). If $\theta(e_2) = 1$, then the SDF graph is not live because $[0\ 1\ 1]^\intercal \xrightarrow{p_3} [2\ 1\ 0]^\intercal \xrightarrow{\text{deadlock}}$.

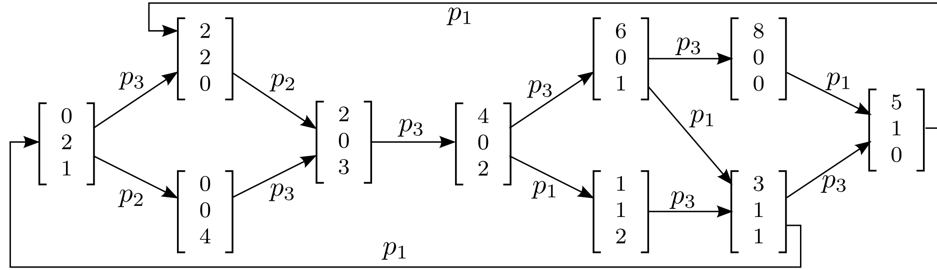


Figure 1.2: Reachability graph of the SDF example.

A Consistency

If a dataflow graph is not consistent, then its execution requires unbounded channels or unbounded number of initial tokens to not deadlock. Only consistent graphs are of interest in this thesis. Consistency is a structural property that can be verified efficiently [17, 117, 41]. A (C|H)SDF graph can be described by a $|E| \times |P|$ topology matrix Γ such that $\forall e_j = (p_i, p_k, v_1^\omega, v_2^\omega) : \Gamma_{j,i} = \frac{\|v_1\|}{|v_1|}, \Gamma_{j,k} = -\frac{\|v_2\|}{|v_2|}$, and $\Gamma_{j,l} = 0$ for any other actor p_l.

For a graph to be consistent, a positive non-null integer solution of the balance equation (Equation 1.1) must exist.

$$\Gamma \vec{r} = \vec{0} \tag{1.1}$$

The minimal solution is called the repetition vector which determines the relative firing frequencies of actors. The repetition vector of the SDF graph in Figure 1.1 is equal to $[2\ 1\ 3]^\intercal$.

The balance equation can be easily explained as follows. Let $|\sigma|_i$ be the number of firings of actor p_i during a complete execution σ. The total number of produced tokens

on channel $e = (p_i, p_k, v_1^\omega, v_2^\omega)$ converges to $\frac{\|v_1\|}{|v_1|}|\sigma|_i$ while the number of consumed tokens converges to $\frac{\|v_2\|}{|v_2|}|\sigma|_k$. A necessary condition for e to be bounded is that

$$\frac{|\sigma|_i}{|\sigma|_k} = \frac{\|v_2\|}{|v_2|}\frac{|v_1|}{\|v_1\|} = \frac{\vec{r}(i)}{\vec{r}(k)}$$

Equation 1.1 ensures that cycles can satisfy this constraint. Any dataflow graph without *undirected* cycles is consistent.

B Graph iteration

An iteration is a sequence of actor firings that returns the graph into its initial marking. The firing sequence must contain $k\vec{r}(i)$ firings of each actor p_i with $k \in \mathbb{N}_{>0}$. Repeating the iteration infinitely results in a bounded and *complete* execution. For the reachability graph in Figure 1.2, the periodic sequence $(p_2\ p_3\ p_3\ p_3\ p_1\ p_1)^\omega$ is such a bounded execution. Not all bounded executions should be periodic. Indeed, the language recognized by the reachability graph in Figure 1.2 (considered as a Büchi automaton) is not periodic.

The balance equation is a necessary but not sufficient condition for the existence of an iteration. In fact, an iteration that returns the graph to its initial marking may not exist. That depends essentially on the initial marking. If the graph is live and consistent, then an iteration always exists [84]. One simple solution to test the existence of an iteration is through simulated execution that consists in constructing the reachability graph [117]. A second solution is to check that all maximal strongly connected components are deadlock-free [84]. The third solution consists in checking that the equivalent HSDF graph of any directed cycle in the CSDF graph is live [41]. A necessary and sufficient condition for a single-rate (i.e. HSDF) graph to be live is that every directed cycle in it contains at least one initial token.

A sufficient and necessary condition for a cycle to be deadlock-free was presented for computation graphs in [109], and adapted for SDF graphs in [141]. A cycle $\{e_2 = (p_1, p_2, a_1^\omega, b_2^\omega), \ldots, e_1 = (p_n, p_1, a_n^\omega, b_1^\omega)\}$ deadlocks if and only if the following integer program has a non-negative integer solution.

$$\Gamma\vec{x} \leq \vec{B}$$

where Γ is the topology matrix of the cycle and $\vec{B}$ is the vector such that $\vec{B}(i) = b_i - \theta(e_i) - 1$. Assume that there is a non-negative integer solution $\vec{x}_0$. After firing each actor p_i as many times as indicated by $\vec{x}_0(i)$, the number of tokens in channel e_i will be less or equal to $\theta(e_i) + \vec{B}(i) = b_i - 1$. This is a deadlocked state.

In the periodic schedule $(p_2\ p_3\ p_3\ p_3\ p_1\ p_1)^\omega$, three instances of actor p_3 are enabled at the same time. Therefore, if there are enough processing units, the three firings can be executed in parallel. This behavior is called auto-concurrency. To constrain the auto-concurrency of an actor p such that at most b firings of actor p can execute in parallel, we need to add a single-rate self-loop $e = (p, p, 1^\omega, 1^\omega)$ such that $\theta(e) = b$. In Figure 1.1, auto-concurrency of actors p_1 and p_3 is disabled in the HSDF graph.

As described for KPNs, to constrain the size of a channel $e = (p_i, p_k, x, y)$ to be less than or equal to b, we may add a feedback channel $e' = (p_k, p_i, y, x)$ such that $\theta(e) + \theta(e') = b$. This way, any bounded graph can be transformed to a strongly connected graph.

A static-order schedule [141] is an infinite sequence of actor firings such that actors in the sequence are fired one after one without overlapping. If the firing sequence is periodic, then the schedule is called a static-periodic schedule (or a cyclic executive).

1.3.2 Timing analysis

In a timed dataflow graph, each actor p_i is associated with a (constant or periodic) worst-case execution time sequence $\eta_i \in \mathbb{N}^\omega$ such that $\eta_i(j)$ is the worst-case execution time of firing $p_i[j]$. As shown in Figure 1.1, worst-case execution times are denoted inside the nodes. When it is needed, a single worst-case execution time of actor p_i is taken as $C_i = \max_j \eta_i(j)$.

One important operational semantics, especially to analyze timing properties, is the self-timed execution [175] which can be described formally in terms of a labeled transition system as proposed in [179, 85].

Definition 1.6 (Self-timed execution)**.** An actor is enabled if there are enough tokens on its input channels. In a self-timed execution, an actor fires as soon as it is enabled.

Figure 1.3 depicts the self-timed execution of the SDF example. At time 10, one instance of p_2 ends leading to channel state $[0\ 0\ 3]^\mathsf{T}$ and hence three instances of actor p_3 start their executions. As illustrated in Figure 1.3 and proved in [85], the state-space of the self-timed execution of a consistent and strongly connected SDF graph consists of a transient phase followed by a periodic phase.

Self-timed boundedness does not coincide with the general notion of (strict) boundedness [84] since self-timed executions are a subset of all the possible executions. All self-timed bounded graphs are bounded but not necessarily strictly bounded; and not all bounded graphs are self-timed bounded.

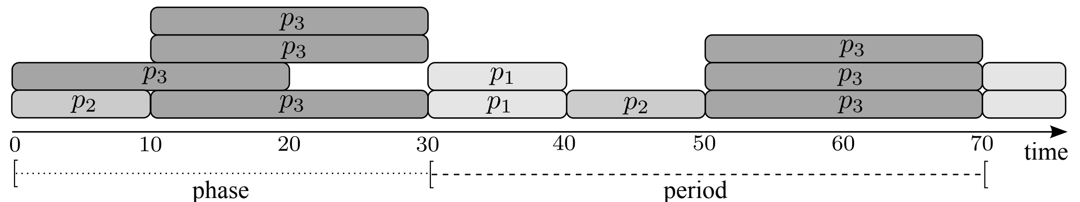


Figure 1.3: Self-timed execution of the SDF example.

A Throughput analysis

One of the most primordial performance metrics of DSP and real-time multimedia applications is throughput. The throughput of an actor p in an execution σ is defined

as the average number of firings of p in σ per unit of time; and it is denoted by $\Theta(p, \sigma)$. The maximum throughput of an actor p is obtained by the self-timed execution; and it is denoted simply by $\Theta(p)$. The self-timed throughput of an actor in a consistent and strongly connected CSDF graph is bounded; However, if an actor is always enabled, then its self-timed throughput tends to infinity. The self-timed throughput of a consistent and strongly connected graph G is defined as $\Theta(G) = \frac{\Theta(p_i)}{\vec{r}(i)}$ for an arbitrary $p_i \in P$.

Throughput analysis has been proposed in the literature either as a (symbolic) state-space exploration [85, 84] or as a maximum cycle mean analysis [175]. In the first approach, the self-timed throughput is computed from the periodic phase of the self-timed execution constructed by a simulated execution. In the second approach, and as proved in [175], the self-timed throughput of the graph is equal to 1 over the maximum cycle mean (MCM) of the equivalent HSDF graph. The MCM of a HSDF graph G, assuming that η_i is a constant sequence for all $p_i \in P$, is defined as

$$\text{MCM}(G) = \max_{\text{cycle } C \in G} \left\{ \frac{\sum\limits_{p_i \in P} C_i}{\sum\limits_{e \in C} \theta(e)} \right\}$$

Efficient algorithms for computing MCM have been proposed in the literature [108, 64]. However, transforming a CSDF graph into a HSDF graph can result in an exponential number of actors compared with the original graph. Because of this, and as illustrated by experimental evaluations [85], state-space exploration approach outperforms the MCM-based approach.

Reduction techniques [80] may help to obtain smaller and hence more analyzable HSDF graphs. One proposed technique is to abstract a (H)SDF graph G with a smaller one G' where actors with identical firing rates are ordered and represented as a single actor. Authors proved that, following their transformation rules, the resulted graph G' will have a conservative throughput; i.e. $\Theta(G') \leq \Theta(G)$. A second technique was a new transformation algorithm of SDF graphs into HSDF graphs that produces not necessarily an equivalent image but only a conservative one. This technique relies on a symbolic state-space exploration (a Max-Plus model) limited to the phase and a period of the self-timed execution. The resulted graph is at most quadratic in the number of initial tokens in the original graph.

As shown in [85, 68], the state-space exploration can be characterized by a Max-Plus algebra [7]. Let $t_i[j]$ be the start time of the jth firing of actor p_i. In Max-Plus algebra, $t_i[j]$ is expressed in terms of the start times of some preceding firings by means of two operators: the maximum $\oplus$ (used in the role of addition) and the ordinary addition $\otimes$ (used in the role of multiplication). For the HSDF example (Figure 1.1(b)), $t_{1,1}[j] = (t_{1,2}[j-1] \oplus 10) \oplus (t_{3,1}[j] \otimes 20) \oplus (t_{3,2}[j] \otimes 20)$. The max-plus time system of a SDF graph can be constructed without transforming the graph into an HSDF graph (see [68] for details). The set of sum-of-products equations can be encoded as a matrix equation

$$\vec{t_j} = A\vec{t}_{j-1}$$

Vector $\vec{t_j}$ describes the start times of the jth firings of all actors; i.e. $\forall p_i : \vec{t_j}(i) = t_i[j]$.

If $\vec{t}_0$ encodes the initial state, then $\vec{t}_j = A^j \vec{t}_0$ describes the progress of the self-timed execution over time. The eigenvalue λ (i.e. the solution of equation $A\vec{t} = \lambda \otimes \vec{t}$) equals the MCM of the graph; and thus the throughput of the graph is equal to $\frac{1}{\lambda}$.

A parametric throughput analysis of SDF graphs is presented in [83] where actors' execution times can be parameters. The throughput is hence computed as function of the parameters. Recalculation of throughput is then simply an evaluation of this function for specific parameter values. This parametric analysis could be useful at design time to avoid recalculation of the throughput for each different configuration.

B Throughput in multiprocessor scheduling

The self-timed execution, under which a maximal throughput can be obtained, assumes that there is an unlimited number of processing units. Multiprocessor scheduling of dataflow graphs assumes a fixed number of processing units. It can be either dynamic or static [114, 175]. The partitioned static-order scheduling problem of dataflow graphs consists in statically (and permanently) mapping actors onto a fixed number of processors $\{A_1, \ldots, A_M\}$ (the *allocation* step), constructing a static-order schedule of each partition A_i (the *scheduling* step), and determining exactly when each actor fires such that all data precedence constraints are met (across processors). This problem has been extensively studied in the two past decades from multiple angles of view. Throughput maximization was one of the most important optimization criteria. Existing solutions either transform the graph into a HSDF graph [175, 176, 139] or compute mapping and scheduling directly on (C)SDF graphs [183, 42].

The achieved throughput depends on both allocation and scheduling. In most approaches the two steps are performed separately as a bin-packing heuristic due to the NP-hardness of the scheduling problem. Firstly, the actors are ordered according to some criterion. In [183], actors are ordered according to their impact on the throughput of the graph. As described before, the throughput of the graph is limited by its critical path. To avoid graph transformation and MCM analysis, authors proposed an approximate cost function to estimate the criticality of an actor p_i.

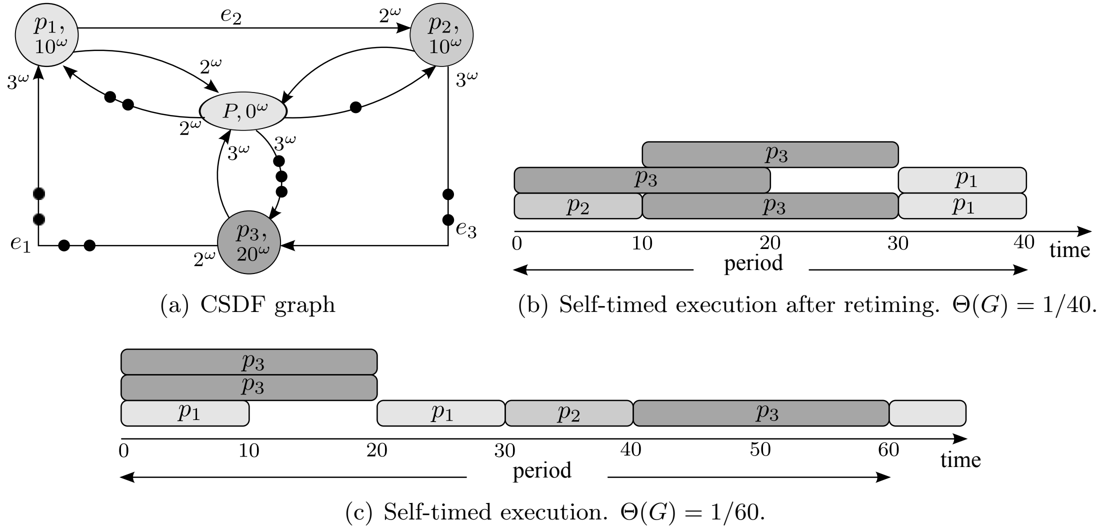
$$\mathrm{cost}(p_i) = \max_{\text{Cycle } C | p_i \in C} \frac{\sum_{p_k \in C} \vec{r}(k) \,.\, \mathrm{avg}(\eta_k)}{\sum_{e \in C} \theta(e)/\tilde{e}}$$

$\mathrm{avg}(\eta_k)$ is the average execution time of actor p_k on different processing units. For a channel $e = (p, q, x, v^\omega)$, value $\tilde{e}$ is equal to $\frac{\|v\|}{|v|}$. After ordering the actors, the second step in the heuristic consists in binding actors, one after another, to processors according to some objective function. In [183, 139], the objective was to balance the load of partitions. The load of a partition is estimated by the processor, memory, and connection utilizations. Actors are then assigned to partitions with a first fit or best fit strategy to achieve the desired objective.

C Retiming technique

Besides the graph topology and execution times, the initial delay distribution may limit the throughput. Retiming is a graph transformation technique for performance optimization (e.g. throughput maximization, minimizing memory usage, decreasing power consumption, etc.) that redistributes the delays and does not affect neither the topology nor the functionality of the graph [128, 199, 198, 191, 148, 122].

Figure 1.4(a) shows the SDF example with a different initial delay distribution ($\theta(e_1) = 4, \theta(e_2) = 0$, and $\theta(e_3) = 2$). We impose that during each period of the self-timed execution, every actor p_i fires only $\vec{r}(i)$ times (we can model this constraint by adding a fictional actor as illustrated in Figure 1.4(a)). As can be deduced from the self-timed execution (Figure 1.4(c)), the self-timed throughput is equal to $\frac{1}{60}$. After retiming the graph ($\theta(e_1) = 0, \theta(e_2) = 2$, and $\theta(e_3) = 1$), the new self-timed throughput is equal to $\frac{1}{40}$ (Figure 1.4(b)).



(a) CSDF graph

(b) Self-timed execution after retiming. $\Theta(G) = 1/40$.

(c) Self-timed execution. $\Theta(G) = 1/60$.

Figure 1.4: Retiming of a SDF graph.

Let G be a consistent and live SDF graph and $\vec{z} : P \longrightarrow \mathbb{Z}$ be a transformation vector such that the new retimed graph is obtained by firing each actor p_i in the original graph $\vec{z}(i)$ times. Hence, for very channel $e = (p_i, p_k, v_1^\omega, v_2^\omega)$, the new number of initial tokens is $\theta'(e) = \theta(e) + \|v_1\|\vec{z}(i) - \|v_2\|\vec{z}(k)$. The retiming vector in the previous example was $[2\ 0\ 1]^\mathsf{T}$. A retiming is legal if it results in nonnegative delay distribution. The retimed graph will be then bounded and live. Most solutions search for an optimal legal retiming or a feasible one that satisfies some throughput constraints.

D Latency analysis

Another useful performance metric of interactive concurrent real-time applications is latency. Latency was defined in [175] for HSDF graphs and consistently generalized in

[86] to SDF graphs. It is measured for two actors p_i and p_k that are usually considered as the input and output of the system. To measure the latency, two fictional actors p_0 and p_{N+1} will be added to the graph such that in each iteration of the graph (according to the repetition vector) these two actors fire only once (i.e. $\vec{r}(0) = \vec{r}(N+1) = 1$). The fictional actors have null execution times (i.e. $\eta_0 = \eta_{N+1} = 0^\omega$) in order to not affect the timing analysis. Two channels are added to the graph: $e = (p_0, p_i, \vec{r}(i)^\omega, 1^\omega)$ and $e' = (p_k, p_{N+1}, 1^\omega, \vec{r}(k)^\omega)$ with $\theta(e) = \theta(e') = 0$. The $\varkappa(j)^{th}$ firing of p_{N+1} is the firing that consumes some tokens whose construction involves the reading of some tokens produced by the jth firing of actor p_0. Hence, the latency of actors p_i and p_k in an execution σ is given as

$$L^\sigma_{i,k} = \max_{j \in \mathbb{N}_{>0}} \{t_{N+1}[\varkappa(j)] - t_0[j]\}$$

where $t_i[j]$ is the start time of firing $p_i[j]$. The minimal latency is obtained by the self-timed execution. However, due to resource constraints, the self-timed execution could not be possible. The worst minimal achievable latency is obtained for a single processor.

1.3.3 Memory analysis

Memory analysis concerns the minimization of both code size and channel sizes. Memory constraints are strong non-functional requirements since the amount of on-chip memory in embedded systems is severely limited due to several constraints (e.g. cost, power consumption, speed penalties, etc.).

A Code size minimization

Automatic code generation is an essential feature of any dataflow-based design environment. One widely used code generation strategy in single-processor scheduling is called *threading* which inlines the code of actors in a static-periodic schedule. One objective of such strategy is to minimize the resulted code size by using loops as efficient as possible. For the schedule $(p_2\ p_3\ p_3\ p_3\ p_1\ p_1)^\omega$ of the SDF example, the generated code is shown in Listing 1.3.

Listing 1.3: Software synthesis from the SDF example.

```
while(true) do{
     code of p2
     for(i=0;i<3;i++){  code of p3  }
     for(i=0;i<2;i++){  code of p1  }
 }
```

As shown in the reachability graph of the SDF example, there is an infinite number of static-periodic schedules that may have different code sizes. Periodic schedules that can be described with a regular expression in which each actor appears only once, and hence have a minimal size, are called single appearance schedules [30, 33]. The previous

example is a single appearance schedule that can be described with $(p_2(p_3)^3(p_1)^2)^\omega$. As has been proved, every consistent acyclic SDF graph has at least one single appearance schedule. More results can be found in [32, 17, 31]. For instance, a SDF graph has a single appearance schedule if and only if each strongly connected component has a single appearance schedule. Furthermore, a consistent strongly connected SDF graph has a single appearance schedule if and only if all its strongly connected subgraphs are loosely interdependent. As an illustration, the SDF example is loosely interdependent because it can be partitioned into two subsets $A = \{p_2\}$ and $B = \{p_1, p_3\}$ such that actors in subset A can fire, as many times as indicated by the repetition vector, before any firing of actors in subset B. If the delay distribution is such that $\forall i : \theta(e_i) = 1$, then the SDF example is not loosely interdependent; a single appearance schedule does not hence exist. Since a graph may have multiple single appearance schedules, selecting the one with the least buffering requirements was the choice of most solutions [143, 145].

B Buffer minimization

Minimizing buffer storage capacity has been the subject of several heuristics and exact approaches in the last two decades. It consists in finding the smallest buffering requirements that allow for a complete execution. The buffer minimization problem is known to be NP-complete [17, 141] by reduction of the Feedback arc set problem to buffer minimization in HSDF graphs.

If channels are implemented as separated storage spaces (i.e. empty space in one channel cannot be used to store tokens of other channels), then the buffer minimization problem consists in finding a schedule that minimizes $\sum_{e \in E} \digamma(e) \max_t\{\theta_t(e)\}$ where $\theta_t(e)$ is the number of tokens in channel e at time t and $\digamma(e)$ is the size of a token (deduced from the data type). From the reachability graph of the SDF example and assuming that all tokens in the graph have the same size, the periodic schedule $(p_3\ p_2\ p_3\ p_1\ p_3\ p_1)^\omega$ results in the smallest buffer size distribution $[4\ 2\ 3]^\intercal$. This distribution is indeed minimal because the lower bound of the size of a channel $e = (p_i, p_k, a^\omega, b^\omega)$ in a SDF graph that can be achieved by any valid schedule is given in [141] by $a + b - \gcd(a, b) + \theta(e) \bmod \gcd(a, b)$ if $0 \leq \theta(e) \leq a + b - \gcd(a, b)$, and by $\theta(e)$ otherwise. This first variation of the buffer minimization problem is considered, for instance, in [132, 91, 63, 1].

If memory is shared between buffers (i.e. all buffers share a single storage space), then the buffer minimization problem consists in finding the schedule that minimizes $\max_t\{\sum_{e \in E} \digamma(e)\theta_t(e)\}$. Unlike the previous variation, this latter approach, considered in [142], must take into account the lifetime of tokens. Hybrid methods may combine the two variations as proposed in [147, 82]. In case the smallest buffer size distribution is not unique, scheduling algorithms choose either the distribution with the highest throughput or the one with the smallest code size.

Another variation of the buffer minimization problem is to minimize the buffering requirements while satisfying a throughput constraint [194, 195, 18]. Indeed, the schedule with minimal buffer sizes may have a throughput that does not satisfy timing constraints. Figure 1.5 shows, for each total amount of buffering requirements, the buffer

size distribution that gives the highest self-timed throughput. For example, if the total storage space is constrained to be at most 10, then there are two buffer size distributions ($[5\ 2\ 3]^\mathsf{T}$ and $[4\ 2\ 4]^\mathsf{T}$) which achieve the best self-timed throughput (0.014). Since the SDF example is strictly bounded, the maximal throughput that can be achieved with a bounded storage distribution is the one computed by the MCM analysis. The set of all trade-offs between the distribution size and the throughput is called the *Pareto space* of throughput and storage trade-offs. A technique that compute either the exact or the approximate Pareto space of (C)SDF graphs was presented in [184, 180].

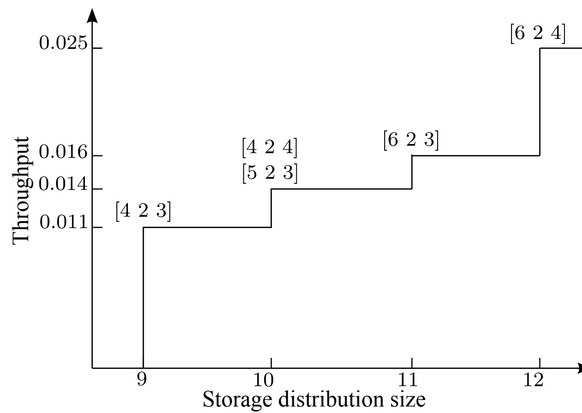


Figure 1.5: Pareto space of the SDF example.

1.4 Real-time scheduling

The inflexibility and difficult maintainability of systems scheduled by cyclic executives (i.e. static-order schedules) led to the development of new real-time scheduling theories such as priority-driven scheduling theory. Much effort has been devoted to identify models of systems, such as the periodic task system model of Lui and Layland [127], from the requirements encountered in the development of real-time systems. Besides models, many theoretical results have been established to predict before the costly implementation phase whether the system will meet its timing requirements even in the worst-case scenarios.

Current models are much mature and general [165, 66, 51, 74]; they consist of periodic, sporadic, and aperiodic tasks (i.e. threads) with implicit deadlines (deadlines equal to periods), constrained deadlines (deadlines less than or equal to periods), or arbitrary deadlines. More requirements are considered such as jitters, resource access protocols, precedence constraints, etc. However, some models are well understood than others. For instance, aperiodic tasks are less predictable than periodic and sporadic tasks. In this section, we will consider a restricted but yet expressive task model that corresponds to our needs for a priority-driven scheduling of dataflow graphs.

1.4.1 System models and terminology

A real-time system comprises a set of N tasks (a task set) $P = \{p_1, p_2, \ldots, p_N\}$, each task consists of an infinite or finite sequence of jobs (invocations, activations, or requests). The jth job of task p_i is denoted by $p_i[j]$. A job executes in a sequential fashion and does not self-suspend. In addition, auto-concurrency is not allowed and tasks are independent and do not share any resources (except the processors of course) unless it is stated otherwise. A run-time scheduler (dispatcher or operating system) controls which job is executing at a given moment on a given processor.

Timing constraints are of a particular interest in real-time computing. In the presented model of computation, timing constraints are expressed in terms of deadlines. If $R_i[j]$ is the release (or arrival) time of job $p_i[j]$ and d_i is the relative deadline of task p_i, then job $p_i[j]$ meets its timing constraint if its execution completes before the absolute deadline $D_i[j] = R_i[j] + d_i$. As we mentioned before, deadlines can be either *hard* or *soft*.

Scheduling theory generally assumes three kinds of tasks, characterized by the arrival pattern of their jobs.

- *Periodic tasks*. Jobs of a periodic task p_i arrive at fixed intervals; i.e. $\forall j \in \mathbb{N}_{>0} : R_i[j+1] = R_i[j] + \pi_i$. The constant π_i is called the period of the task. This arrival pattern is thus time-driven. A periodic task p_i is characterized by a worst-case execution time C_i, a *hard* deadline d_i, and a first release time (called phase or offset) $r_i = R_i[1]$.

- *Aperiodic tasks*. Jobs of an aperiodic task arrive randomly, usually in response to some external triggering event. This arrival pattern is thus event-driven. Besides having unknown interarrival time of requests, aperiodic tasks have unknown (or extremely large) worst-case execution times. Due to this unpredictability, aperiodic tasks can have only soft deadlines.

- *Sporadic tasks*. A sporadic task p_i has a fixed minimum interarrival time π_i; i.e. $\forall j \in \mathbb{N}_{>0} : R_i[j+1] \geq R_i[j] + \pi_i$. The worst-case behavior is usually considered when analyzing sporadic tasks. Hence, they can be considered just as periodic tasks.

A periodic task set, whose all tasks are periodic, can be classified either as *synchronous* or *asynchronous*. A task set is synchronous if there is some point in time at which all tasks arrive simultaneously (i.e. $\exists t : \forall p_i \in P : \exists j \in \mathbb{N}_{>0} : R_i[j] = t$); and asynchronous otherwise. Furthermore, it can be classified as an *implicit* task set ($\forall p_i \in P : d_i = \pi_i$), as a *constrained* task set ($\forall p_i \in P : d_i \leq \pi_i$) , or as an *arbitrary* task set ($\forall p_i \in P : d_i \lesseqgtr \pi_i$). A task set is also characterized by its utilization $U = \sum\limits_{p_i \in P} U_i$ with $U_i = \frac{C_i}{\pi_i}$, its density $\mu = \sum\limits_{p_i \in P} \mu_i$ with $\mu_i = \frac{C_i}{\min\{d_i, \pi_i\}}$, and its hyperperiod $H = \mathrm{lcm}\{\pi_i | p_i \in P\}$.

A task set is said to be *feasible* w.r.t. a given system if there is a scheduling algorithm under which all the possible sequences of jobs that may be generated by the task set are scheduled on that system without missing any deadline. A scheduling algorithm is

said to be *optimal* w.r.t. a system and a task model if it can schedule any feasible task set that conforms to the task model.

A task set is said to be *schedulable* according to a given scheduling algorithm if all its tasks do never miss their deadlines under that algorithm. If all the task sets that are qualified as schedulable by a schedulability test w.r.t. a scheduling algorithm are in fact schedulable, then the schedulability test is said to be *sufficient*. If all the task sets that are deemed unschedulable by the test are in fact unschedulable, then the test is referred to as *necessary*. An *exact* schedulability test is one that is sufficient and necessary.

For implicit-deadline periodic task sets, the *utilization bound* (UB) U_A of a scheduling algorithm A is defined as the minimum utilization of any task set that is only just A-schedulable; i.e. any task set with $U \leq U_A$ is A-schedulable.

A Uniprocessor scheduling algorithms

The scheduling algorithm selects, at each decision instant, a single job among all the active jobs which are waiting for the processor in order to start or continue their executions. If there is no active job, the processor is said to be *idle*. Most of scheduling algorithms do not idle the processor unless there is no active jobs; they are called *work-conserving* algorithms. One of the most important class of dynamic scheduling algorithms is the class of priority-driven scheduling policies. In priority-driven scheduling, each job $p_i[j]$ is assigned a priority $\omega_i[j] \in \mathbb{N}_{>0}$ according to some policy with 1 being the highest priority. At each decision instant, the scheduler chooses the active job with highest priority to execute.

Based on the set of instants at which a scheduler must select an active job, algorithms can be classified as preemptive, nonpreemptive, or cooperative.

- *Preemptive*. Tasks can be preempted by a higher priority task at any time. Hence, schedule decisions are taken when jobs complete, or when jobs arrive.

- *Nonpreemptive*. Once a task starts executing, it cannot be preempted. Hence, schedule decisions are taken when jobs complete, or when one or more new jobs arrive while the processor is idle.

- *Cooperative*. Tasks can only be preempted at specific points within their execution.

In the sequel, all the presented algorithms are fully-preemptive. Based on the priority assignment strategy, scheduling algorithms can be furthermore classified as fixed-priority or dynamic-priority.

Fixed-priority scheduling In FP scheduling, all jobs of a task p_i have the same fixed priority w_i. Priorities are assigned to tasks at compile-time and do not change once the application starts executing. Priorities can be assigned arbitrarily by the designer, or according to some priority assignment policy. Two well-known priority assignment policies are the *Rate Monotonic* (RM) policy [127, 46] and the *Deadline Monotonic* (DM)

policy [124]. In RM policy, tasks with shorter periods are associated with higher priorities; while in DM policy, tasks with shorter deadlines are assigned higher priorities. Clearly, for an implicit-deadline task set, the DM policy reduces to the RM policy.

For synchronous constrained periodic task sets, the DM policy is proven to be optimal [127, 124] among all FP assignments. The complexity of ordering tasks according to DM policy is equal to $\mathcal{O}(N \log_2 N)$. The DM policy is however not optimal for asynchronous task sets [124]. The optimal priority assignment policy for asynchronous constrained periodic task sets was proposed in [4] and it has a complexity of $\mathcal{O}(E(N^2 + N)/2)$ where E represents the (non-polynomial) complexity of the schedulability test.

Dynamic-priority scheduling In dynamic-priority scheduling, algorithms determine the priorities of each task at run-time. Two well-known methods of dynamic priority assignment are the *Earliest Deadline First* (EDF) policy [127] and the *Least Laxity First* (LLF) policy [138]. In EDF policy, the job with the earliest absolute deadline among all active jobs is given the highest priority; while in LLF policy, the job with the smallest laxity among all active jobs is given the highest priority. The laxity of a job is the difference between its upcoming deadline and its (estimated) remaining computation time. Both EDF and LLF are proved to be optimal; however LLF has more context switching overhead than EDF. For this reason, most of the proposed theoretical results concern the EDF policy.

B Multiprocessor scheduling algorithms

Multiprocessor scheduling is a much more difficult problem than uniprocessor scheduling and only few of the results obtained for a single processor can be directly generalized to the multiple processor case. Multiprocessor systems can be classified into three categories [66]:

- *Homogeneous*. The system comprises M identical processors. A task has hence the same rate of execution on all processors. The presented algorithms in this section are tailored to this architecture unless it is stated otherwise.

- *Uniform*. Processors may have different speeds. Hence, a processor of speed 2 will execute all tasks at exactly twice the rate of a processor of speed 1.

- *Heterogeneous*. Processors are different. Some tasks may not be able to execute on some processors. Hence, the rate of execution of a task depends both on the processor and the task.

Besides the priority assignment problem described for uniprocessor scheduling, multiprocessor scheduling has to solve an allocation problem: on which processor a task should execute at each instant. As a result of preemption, a task may migrate from one processor to another. Based on migration, scheduling algorithms can be classified as:

- *Partitioned*. Each task is permanently allocated to a processor and no migration is permitted. Partitioned algorithms are hence not work-conserving; a processor can be idle while some active tasks are waiting for another processor.

- *Global*. A single job can migrate to and execute on different processors.

The current state of the art favors the partitioned scheduling over the global one for the following advantages of partitioned scheduling: (1) There is no penalty in terms of migration cost. (2) A task that overruns its time budget can only affect other tasks on the same processor. (3) Once an allocation of tasks has been achieved, well-mature scheduling techniques for uniprocessor systems can be used. The main drawback of partitioned scheduling is that the allocation problem is analogous to the bin packing problem which is known to be NP-hard. As stated for the multiprocessor static scheduling of dataflow graphs, most of solutions use some bin packing heuristics such as first fit, next fit, best fit, and worst fit. Besides this reason, the recent advancement in multiprocessor technology that reduces the migration penalties has rekindled the interest in global scheduling.

1.4.2 EDF schedulability analysis

This section reviews some existing EDF schedulability tests that will be used in this thesis. In the sequel, we will focus more on periodic task sets with implicit or constrained deadlines. One of the outstanding results about EDF it is its optimality for uniprocessor systems. Unfortunately, EDF scheduling is not optimal for multiprocessor systems.

A Uniprocessor scheduling

Liu and Layland [127] proved that an implicit-deadline periodic task set is schedulable if and only if

$$U \leq 1 \tag{1.2}$$

The UB of the EDF policy U_{EDF} is hence equal to 1. This is an exact test that has a complexity of $\mathcal{O}(N)$. For arbitrary periodic task sets, the previous test is only a necessary test; while $\mu \leq 1$ is only a sufficient condition [131]. Devi [71] proposed a sufficient schedulability test of arbitrary periodic task sets. Assuming that the task set $\{p_1, \ldots, p_N\}$ is arranged in order of non-decreasing relative deadlines, the task set is EDF-schedulable if

$$\forall 1 \leq k \leq N : \sum_{i=1}^{k} U_i + \frac{1}{d_k} \sum_{i=1}^{k} \left(\frac{\pi_i - \min\{\pi_i, d_i\}}{\pi_i} \right) .C_i \leq 1 \tag{1.3}$$

This test is better than the density condition, but has a complexity of $\mathcal{O}(N \log N)$ because it requires sorted task sets. An improved sufficient test that has a complexity of $\mathcal{O}(N^2)$ was presented in [137].

Exact schedulability analysis of constrained periodic task sets is known to be Co-NP-hard [73]. In [123], it was noted that a periodic task set is schedulable if and only

if all absolute deadlines in the interval $[0, L]$ are met where $L = \max_{p_i \in P}\{r_i\} + 2H$ (recall that H is the hyperperiod). The upper bound L is called a *feasibility bound*. This exact test has however an exponential complexity.

A pseudo-polynomial-time algorithm based on the processor demand function has been proposed for constrained periodic task systems. The processor demand function for a task p_i is a function on time interval $[t_1, t_2]$ that gives the amount of computation needed by all jobs of p_i that have both their arrival time and their deadlines within the interval $[t_1, t_2]$; i.e. $h_i(t_1, t_2) = \sum_{R_i[j] \geq t_1 \wedge D_i[j] \leq t_2} C_i$. The processor demand of the entire task system is hence equal to $h(t_1, t_2) = \sum_{p_i \in P} h_i(t_1, t_2)$. A task set is EDF-schedulable if and only if $\forall t_1, t_2 : h(t_1, t_2) \leq t_2 - t_1$. Let us take $h(t)$ to be the maximum processor demand over a contiguous interval of length t, hence $h_i(t)$, for synchronous task sets, is given as

$$h_i(t) = \max\{0, 1 + \left\lfloor \frac{t - d_i}{\pi_i} \right\rfloor\} \tag{1.4}$$

Theorem 1.1 ([14, 15]). *A synchronous arbitrary periodic task set is EDF-schedulable if and only if* $U \leq 1$ *and* $\forall t < L : h(t) \leq t$ *where* $L = \max\{d_1, \ldots, d_N, \max_{p_i \in P} \{\pi_i - d_i\} \frac{U}{1-U}\}$.

This schedulability test is only sufficient for asynchronous task sets. A better feasibility bound for constrained systems was proposed in [159] as

$$L = \frac{\sum_{p_i \in P} (\pi_i - d_i) U_i}{1 - U} \tag{1.5}$$

Another feasibility bound is the length of the *synchronous busy period* [159]. The busy period starts when all tasks are released simultaneously at their maximum rate, and ends by the first processor idle item. The length of the busy period can be computed by the following recurrence.

$$w^0 = \sum_{p_i \in P} C_i \qquad\qquad w^{m+1} = \sum_{p_i \in P} \left\lceil \frac{w^m}{\pi_i} \right\rceil C_i \tag{1.6}$$

When the recurrence stops (i.e. $w^{m+1} = w^m$), then $L = w^m$.

The processor demand test ($h(t) \leq t$) needs to be checked only at the absolute deadlines in the interval $[0, L]$ since the processor demand does not change from one point t_1 to another point t_2 unless there is at least one absolute deadline between the two points. This testing set can be however very large. The Quick convergence Processor-demand Analysis (QPA) was proposed in [197, 196] in order to reduce the testing set. Instead of checking all deadlines in the increasing order, the proposed algorithm starts from the last deadline in the testing set and moves backward, skipping many intermediate deadlines thanks to the following lemma, where d, d', d_m, and d^* denote some absolute deadlines.

Algorithm 1: QPA algorithm

$t = d_m$;
while $h(t) \leq t \wedge h(t) > \min\{d\}$ **do**
 if $h(t) < t$ **then** $t = h(t)$; **else** $t = \max\{d | d < t\}$;
if $h(t) \leq \min\{d\}$ **then** the task set is schedulable;
else the task set is not schedulable;

Lemma 1.1 ([196]). *For an unschedulable task set, if $h(d_m) \leq d_m$, then $d^* < h(d^*) \leq d'$, where $d_m = \max\{d |\ d \leq L\}, d^* = \max\{d | 0 < d < L \wedge h(d) > d\}$, and $d' = \min\{d | d > d^*\}$.*

From Lemma 1.1, it is easy to deduce that $\forall t \in [h(d^*), L] : h(t) \leq t$. Based on this result, Listing 1 represents the QPA algorithm (from [197]).

B Multiprocessor scheduling

Regardless of the scheduling algorithm, an implicit-deadline periodic task set is *feasible* on M processors if and only if $U \leq M$ and $\forall p_i \in P : U_i \leq 1$ [98]. A constrained periodic task set is feasible only if $\max_t\{\frac{h(t)}{t}\} \leq M$ [12].

For partitioned EDF scheduling, several heuristic allocation algorithms have been proposed. Let $(V_j)_{j=1..M}$ be the set of M initially empty partitions (V_j consists of all the tasks allocated to the jth processor) and let $U^j = \sum_{p_i \in V_j} U_i$ be the utilization of partition V_j. For an implicit-deadline task set and using a first fit allocation strategy, each task p_i (tasks are ordered by decreasing utilization) is assigned to the first processor V_j with enough capacity (i.e. $\{p_i\} \cup V_j$ is EDF-schedulable on the jth processor; hence $U_i + U^j \leq 1$). It has been shown in [129] that for any reasonable allocation algorithm the UB of the scheduling algorithm is bounded by

$$M - (M-1)U^* \leq U_{\text{EDF}} \leq \frac{\left\lfloor \frac{1}{U^*} \right\rfloor M + 1}{\left\lfloor \frac{1}{U^*} \right\rfloor + 1} \quad (1.7)$$

where $U^* = \max_{p_i \in P}\{U_i\}$. EDF with a best fit allocation strategy can achieve the highest UB.

As for uniprocessor systems, the processor demand approach is usually used for partitioned EDF scheduling of constrained task sets. Such a technique has been proposed in [13]. Firstly tasks are arranged in non-decreasing order of their relative deadlines (i.e. a DM ordering). Task p_i is assigned to the first partition V_j that satisfies

$$d_i - \sum_{p_k \in V_j} h_k^*(d_i) \geq C_i \quad (1.8)$$

Where $h_i^*(t)$ is an upper bound of the processor demand $h_i(t)$. Simply, $h_i^*(t) = C_i + U_i(t - d_i)$ if $t \geq d_i$, and 0 otherwise. This scheduling heuristic has a polynomial-time complexity.

Concerning global EDF scheduling of implicit-deadline periodic task sets, the UB U_{EDF} is equal to $M-(M-1)U^*$ [88]. A constrained task set is schedulable if $\mu \leq M-(M-1)\mu^*$ where $\mu^* = \max_{p_i \in P}\{\mu_i\}$. A theoretical and experimental comparison between global EDF schedulability tests can be found in [25]. One particular test, that will be used in this thesis, is the one based on the forced-forward demand bound function.

Theorem 1.2 ([25, 11])**.** *A constrained periodic task set is schedulable if* $\exists\gamma : \mu^* \leq \gamma < \frac{M-U}{M-1} - \epsilon$ *(with an arbitrary small* ϵ*) such that* $\forall t \leq L : h(t,\gamma) \leq t$ *where*

$L = \frac{\sum_{p_i \in P}(\pi_i - d_i)U_i}{M-(M-1)\gamma-U}$ and $h(t,\gamma) = \frac{\mathit{ffdbf}(t,\gamma)}{M-(M-1)\gamma}$. We have that $\mathit{ffdbf}(t,\gamma) = \sum_{p_i \in P} \mathit{ffdbf}_i(t,\gamma)$ is the forced-forward demand bound function; and we have that: $z_i = (t \mod \pi_i)$ and

$$\mathit{ffdbf}_i(t,\gamma) = \left\lfloor \frac{t}{\pi_i} \right\rfloor C_i + \begin{cases} C_i & \text{if } z_i \geq d_i \\ C_i - (d_i - z_i)\gamma & \text{if } d_i > z_i \geq d_i - \frac{C_i}{\gamma} \\ 0 & \text{otherwise} \end{cases}$$

It is sufficient to check the condition $h(t,\gamma) \leq t$ only at absolute deadlines in the interval $[0, L]$. Furthermore, the QPA technique can be used to reduce the testing set. Listing 2 represents the QPA-FFDBF algorithm proposed in [25].

Algorithm 2: QPA-FFDBF algorithm

$\gamma = \mu^*$;
while $\gamma < \frac{M-U}{M-1}$ **do**
  $t = L$;
  while $\min\{d\} < h(t,\gamma) \leq t$ **do**
    $t = \min\{h(t,\gamma), \max\{d | d < t\}\}$;
  if $h(t,\gamma) \leq \min\{d\}$ **then return** the task set is schedulable ;
  $\gamma = \gamma + \epsilon$;
return the task set is unschedulable;

An experimental comparison of partitioned and global EDF scheduling techniques can be found in [8].

C EDF with precedence constraints

In many hard-real time systems, communication among tasks should be deterministic. One approach to achieve this goal is to model communication requirements as precedence constraints. Let J_1 and J_2 be two jobs such that each job J_i has an absolute deadline d_i, a release time r_i, and worst-case execution time C_i. A precedence constraint $J_1 \rightarrow J_2$ implies that J_2 starts executing only after J_1 ends. An algorithm was proposed in [58] that solves this problem (for uniprocessor systems) by transforming

dependent jobs (i.e. adjusting their release times and deadlines) into new *independent* jobs such that EDF scheduling of the new set of jobs satisfies the precedence constraints.

Since J_2 cannot start before the completion of J_1, it is safe to change its release time to be $r_2 = \max\{r_2, r_1 + C_1\}$. Note that if there is a precedence constraint $J_3 \to J_1$, then r_1 must be adjusted before computing the new r_2. The second step in the algorithm is to adjust the deadlines. Since J_2 must start executing at most at $d_2 - C_2$ (otherwise it misses its deadline), it is safe to adjust the deadline of J_1 to be $d_1 = \min\{d_1, d_2 - C_2\}$. This way, the new J_2 will not preempt the new J_1.

This transformation algorithm can be implemented in $\mathcal{O}(n)$. It also ensures that if the new job set is EDF-schedulable, then the original set under precedence constraints is also EDF-schedulable.

1.4.3 Fixed-priority schedulability analysis

As with EDF policy, we will consider both uniprocessor and multiprocessor FP scheduling. We will assume that each task can have a distinguished priority.

A Uniprocessor scheduling

An implicit-deadline periodic task set is RM-schedulable if $U \leq U_{RM} = N(\sqrt[N]{2} - 1)$ [127]. This is only a sufficient schedulability test. The UB depends on the number of tasks in the system and we have that $\lim_{N \to \infty} U_{RM} = \ln 2$. A less pessimistic sufficient test was presented in [37] as $\prod_{i=1}^{N} (U_i + 1) \leq 2$. For constrained task sets $\mu \leq U_{RM}$ is a sufficient DM schedulability test.

The exact schedulability test of synchronous constrained periodic task sets, regardless of the priority assignment policy, is based on the notion of worst-case response times [3, 103]. A task set is schedulable if and only if the worst-case response time R_i of each task p_i is less than or equal to its deadline; i.e. $\forall p_i \in P : R_i \leq d_i$. The worst-case response time R_i is the solution of the following recurrence.

$$R_i^0 = C_i \qquad R_i^{m+1} = C_i + \sum_{w_k < w_i} \left\lceil \frac{R_i^m}{\pi_k} \right\rceil C_k \tag{1.9}$$

The response time analysis (RTA) has a pseudo-polynomial complexity. Further improvement can be achieved by reducing the number of iterations required to solve the recurrence equation [130, 67]. Efficient (polynomial time) computation of *approximate* response times can be desirable especially for online admission tests. Assume that R_i is bounded by $R_i^l \leq R_i \leq R_i^u$, then if $\forall p_i \in P : R_i^u \leq d_i$, then the task set is schedulable. Dually, if $\exists p_i \in P : R_i^l > d_i$, then the task set is unschedulable. As noticed in [166], we have that $x \leq \lceil x \rceil < x + 1$. Thus, $R_i^l = \frac{C_i}{1 - \sum_{w_k < w_i} U_k}$ and $R_i^u = \frac{C_i + \sum_{w_k < w_i} C_k}{1 - \sum_{w_k < w_i} U_k}$. A better

upper bound is proposed in [40] as

$$R_i^u = \frac{C_i + \sum\limits_{w_k < w_i} C_k(1 - U_k)}{1 - \sum\limits_{w_k < w_i} U_k} \tag{1.10}$$

B Multiprocessor scheduling

The UB of any FP partitioning algorithm of implicit-deadline periodic task sets is upper bounded by $\frac{M+1}{\sqrt[M+1]{2}+1}$ [144]. An allocation strategy, that has a UB equal to $(M-2)(1-U^*) + 1 - \ln 2$, was proposed in [126]. It attempts to allocate tasks with harmonic periods (i.e. periods that are close to harmonics of each other) to the same processor. Tasks on each processor are then RM-scheduled. In a first fit strategy, a task p_i (tasks ordered according to their RM priorities) is assigned to the first partition V_j that satisfies $U_i + U^j \leq U_{RM}$. The UB of such strategy is equal to $(M-1)(\sqrt{2}-1) + (N-M+1)(\sqrt[N-M+1]{2}-1)$ [126].

A partitioned DM scheduling algorithm of constrained periodic task sets is proposed in [75]. It orders tasks according to their DM priorities, and then assigns each task p_i to the first partition V_j that satisfies $d_i - \sum\limits_{p_k \in V_j} (C_k + U_k d_i) \geq C_i$. Notice that this approximate condition is not just more than $R_i^u \leq d_i$ with $R_i^u = \frac{C_i + \sum\limits_{p_k \in V_j} C_k}{1 - \sum\limits_{p_k \in V_j} U_k}$. However, we have shown that there is a better upper bound. When a task is assigned to a partition, the worst-case response times of the already assigned tasks do not change because they have higher priorities than the new task.

The UB of any global fixed-priority scheduling algorithm of implicit-deadline task sets, where priorities are defined as a scale-invariant function of tasks periods and worst-case execution times, is upper bounded by $(\sqrt{2}-1)M$. The UB of global RM algorithm is equal to $\frac{M}{2}(1-U^*) + U^*$ [27]. Furthermore, any implicit-deadline task set with $U^* \leq \frac{M}{3M-2}$ and $U \leq \frac{M^2}{3M-1}$ is global RM-schedulable [2]. A sufficient response time analysis for global FP scheduling of constrained periodic task sets has been proposed in [26]. A task set is schedulable if $\forall p_i \in P : R_i \leq d_i$ where R_i is an upper bound on the response time of task p_i computed by the following recurrence.

$$R_i = C_i + \left\lfloor \frac{1}{M} \sum_{w_k < w_i} I_k(R_i) \right\rfloor \tag{1.11}$$

Where $I_k(R_i)$ is an upper bound on the interference due to task p_k within the worst-case response time of p_i and which is computed as follows.

$$I_k(R_i) = \min\{R_i - C_i + 1, N_k C_k + \min\{C_k, R_i + d_k - C_k - N_k \pi_k\}\} \tag{1.12}$$

$$N_k = \left\lfloor \frac{R_i + d_k - C_k}{\pi_k} \right\rfloor \tag{1.13}$$

More global FP schedulability tests can be found in [66].

C Aperiodic servers

In this section, we will consider uniprocessor FP scheduling of systems that consist of a set of hard periodic tasks P and a set of soft aperiodic tasks P'. Scheduling aperiodic tasks based on their priorities may cause some lower priority hard tasks to miss their deadlines. One simple solution to prevent soft aperiodic tasks from interfering with hard periodic tasks is to execute them as background tasks; i.e. they execute only when there are no ready periodic jobs. However, this solution generally leads to long response time of aperiodic tasks. In the past decades, many techniques based on aperiodic servers have been devised to improve the average response time of soft aperiodic tasks; examples of such techniques are: polling servers [121], deferrable servers [178], sporadic servers [174], priority exchange servers [173], etc. An aperiodic job executes as the capacity of its server is not exhausted, then it waits for the next replenishment of the server capacity according to its replenishment period and strategy.

Assume that the system contains one server p_s with a capacity C_s and a replenishment period π_s. The server has generally the highest priority. One important property of polling, deferrable, and sporadic servers is that they can be considered just as sporadic tasks (sometimes with jitters) in the response time analysis. Sporadic servers are often considered the best because they achieve a higher processor utilization and can be considered in the schedulability analysis just like sporadic tasks [174]. Bernat and Burns have shown that there is no big difference between the performance of a deferrable server and a sporadic one [22]. They have also shown that a deferrable server can be also considered in the schedulability analysis as a sporadic task with a jitter. Hence, the worst-case response time of a periodic task can be computed as

$$R_i = C_i + \sum_{w_k < w_i} \left\lceil \frac{R_i}{\pi_k} \right\rceil C_k + (1 + \left\lceil \frac{R_i - C_s}{\pi_s} \right\rceil) C_s \tag{1.14}$$

Parameters of servers that minimize the average response time are generally obtained by simulation. Some selection criteria have been proposed in [22]. If the system contains multiple servers, then a capacity sharing protocol like the one described in [23] may increase the responsiveness of aperiodic tasks.

1.4.4 Symbolic schedulability analysis

Designing real-time systems is a very complicated task due to the large number of parameters to be considered by the designer such as: task priorities, deadlines, phases, etc. Parametric analysis is a powerful tool to explore the space of design parameters and find hence the *optimal* values or prune non-feasible solutions from the design space. Symbolic (or parametric) schedulability analysis consists in finding the schedulability region; i.e. the set of values of parameters for which the system is schedulable when using a specific scheduling policy. Optimization objectives, such as maximizing the processor utilization, can be further considered. Few works, compared to the standard schedulability analysis, have addressed this problem and they target mainly FP scheduling for single processor systems.

In [35], symbolic FP schedulability analysis of constrained periodic task sets is considered. Periods, deadlines, and priorities are assumed to have known values; while worst-case execution times are considered as free variables. The proposed technique computes the C-space such that the system is FP schedulable at each point $(C_1, \ldots, C_N)$ in that space. The C-space is constructed by deducing linear constraints on C_i using Lehoczky schedulability test [120]. A synchronous task set is schedulable if and only if

$$\bigwedge_{i=1\ldots N} \bigvee_{t \in S_i} \sum_{w_k \leq w_i} \left\lceil \frac{t}{\pi_k} \right\rceil C_k \leq t \tag{1.15}$$

where $S_i = \bigcup_{w_k \leq w_i} \{r\pi_k | r = 1 \ldots \left\lceil \frac{\pi_i}{\pi_k} \right\rceil\}$. The schedulability region is therefore the union of some convex regions; hence the performance optimization problem can be solved using convex optimization techniques.

Both periods and execution times are considered as parameters in the sensitivity analysis proposed in [39, 34]. Assuming a FP scheduling policy with known priorities, the proposed analysis computes either the C-space (execution times are the only free variables) or the f-space (periods are the only free variables). Construction of the f-space is very important in control systems domain, where control tasks are known and their activation rates needed to be computed. For a given unschedulable task set, the sensitivity analysis computes the minimal modifications on either periods or execution times that bring the system into the schedulability region. The f-space is constructed by deducing constraints on π_i using the schedulability test provided in [163, 38]. Assuming that tasks are ordered according to their priorities (i.e. $\forall p_i : w_i = i$), a task p_i in a synchronous implicit-deadline periodic task set is FP-schedulable if and only if $\exists n_1, \ldots, n_{i-1} \in \mathbb{N}_{>0}$ such that

$$C_i + \sum_{j<i} n_j C_j \leq \pi_i \qquad \wedge \qquad \forall k < i : (n_k - 1)\pi_k \leq C_i + \sum_{j<i} n_j C_j \leq n_k \pi_k \tag{1.16}$$

The schedulability region of each task p_i is the union of parallelepipeds, each one resulting from a different selection of tuple $(n_1, \ldots, n_{i-1})$. The schedulability region of the task set is hence the intersection of the schedulability regions of all tasks. Given a cost function F that provides a measure of the application performance, authors proposed an algorithm that searches for the periods that achieve the highest performance. However, the function $F(x)$ must be convex and $\forall i : \frac{\partial F}{\partial \pi_i} \leq 0$; i.e. an increase of any task's rate results in an improvement of the application performance. The algorithm starts by computing an admissible solution, then enumerating, using a branch and bound approach, all the vertices in a direction indicated by the gradient of the cost function. Examples of cost functions are: maximization of the utilization U, minimization of the energy given by $\sum \alpha_i e^{\frac{-\beta_i}{\pi_i}}$, etc.

The previous approach was extended in [185] for task sets with jitters and deadlines as free parameters. In [89], a method was provided to find the optimal priority assignment (i.e. priorities are the only free variables) given the cost function and all the timing characteristics.

Symbolic model checking of parametric timed automata (PTA) has been recently used to perform symbolic schedulability analysis [60, 112, 185, 158, 77]. PTA extend timed automata with parameters that can be used in guards and invariants. The basic idea of this approach is to model the schedulability problem using PTA, and then using existing model checking tools to compute the schedulability region. Thanks to the expressivity of the PTA model, it is possible to perform parametric analysis on a large class of constraints. However, this expressivity comes at the price of a huge complexity.

1.4.5 Real-time scheduling of dataflow graphs

Using real-time scheduling policies to implement dataflow graphs has been the subject of only few works. Unlike in cyclic executives (i.e. static-periodic schedules), each actor is mapped to a periodic real-time task; hence the schedule is strictly periodic (as opposite to static-periodic). One advantage of this scheduling approach is that existing real-time scheduling theories can be used to decide the schedulability of a dataflow specification on a given architecture.

In [151], a system is described as a set of independent periodic tasks (with user-provided periods) that may communicate through sample-and-hold mechanisms that do not require synchronization. The periodic task set is scheduled according to a RM policy. Each task is modeled as a SDF graph where each actor is called a subtask. Subtasks within a task are executed according to a generated static-periodic schedule where a subtask cannot be preempted. Non-preemptive scheduling is known to be NP-hard in the strong sense even for single processor systems [101]. To handle non-preemption, authors have considered the processor as a shared resource and then used existing analysis for priority inheritance protocols.

Real-time scheduling of acyclic computation graphs (with a chain topology) was studied in [87]. The author did not however use a periodic task model but he has used the *Rate-Based Execution* (RBE) model [100] where each task p_i has four parameters (x_i, y_i, d_i, C_i) with x_i is the number of execution of p_i in an interval of length y_i. Jobs are executed using a preemptive EDF scheduling policy. The schedulability analysis ensures that any job $p_i[j]$ released at time $r_i[j]$ will finish before its absolute deadline $D_i[j]$ given as

$$D_i[j] = \begin{cases} r_i[j] + d_i & \textit{if}\ 1 \leq j \leq x_i \\ \max\{r_i[j] + d_i, D_i[j - x_i] + y_i\} & \textit{if}\ j > x_i \end{cases}$$

The task set is schedulable if and only if $\forall l > 0, l \geq \sum\limits_{p_i \in P} \max\{0, \left\lfloor \frac{l-d_i+y_i}{y_i} \right\rfloor x_i C_i\}$. So, each actor in the computation graph is mapped to a task; and a task is activated only when the number of accumulated tokens in a channel exceeds its threshold (i.e. a data-driven execution). The author assumes that both the parameter y of the first actor in the chain and the relative deadlines are user provided. Hence, parameters x and y of all the actors can be deduced using the repetition vector. The author also proposed an analysis to compute the size of buffers.

The scheduling approach proposed in [9, 10] is closely related to our work. In [9], authors have used the implicit-deadline periodic task model to implement acyclic weakly connected CSDF graphs. Firstly, periods are expressed in terms of the period of one actor using the repetition vector; i.e. $\vec{r}(i)p_i = \vec{r}(j)p_j$. Then, the minimum period $p_i^{\min}$ of each actor p_i is computed (assuming unlimited processing units) as follows.

$$p_i^{\min} = \frac{H}{\vec{r}(i)} \left\lceil \frac{\max_{p_k \in P} C_k \vec{r}(k)}{H} \right\rceil \text{ such that } H = \text{lcm}\{\vec{r}(1), \ldots, \vec{r}(N)\} \tag{1.17}$$

Secondly, the earliest start times of actors are computed according to the initial tokens in the channels. Knowing all the timing parameters and assuming unlimited resources, it is easy to compute the minimum buffer sizes. The last step in this approach consists in using the utilization bound of any multiprocessor scheduling algorithm (i.e. using the test $\mu \leq M$) to compute the necessary number of processors. Authors have shown that this approach gives the self-time throughput of a class of graphs called *matched input output rates* graphs. They are graphs satisfying $\max_{p_k \in P} C_k \vec{r}(k) \bmod H = 0$. In [10], authors showed that the implicit-deadline task model does not give the minimum latency (i.e. a self-timed latency) for a class of graphs called unbalanced graphs. A balanced graph must satisfy $\forall p_i, p_k : \vec{r}(i)C_i = \vec{r}(k)C_k$. Authors proposed a technique that constrains the deadlines to obtain a better latency.

1.5 Real-time calculus

Real-time calculus (RTC) [189, 56] is a specialization of network calculus [57] to the domain of real-time and embedded systems. RTC is used for system-level performance analysis of stream processing systems with timing constraints. In RTC, event streams and services offered by resources are modeled in a coherent way. For a given stream s, the number of arrived events in each time interval is lower bounded and upper bounded by two right-continuous, non-negative, subadditive functions s^l and s^u, respectively; i.e. if $s[t_1, t_2)$ is the number of arrived events in interval $[t_1, t_2)$, then $\forall t_1 < t_2 : s^l(t_2 - t_1) \leq s[t_1, t_2) \leq s^u(t_2 - t_1)$. Hence, $s^l(t)$ and $s^u(t)$ can be considered as the minimum and maximum number of events arriving within any interval of length t, respectively. A periodic event (e.g. release of a periodic task) of period π can be bounded by two staircase functions of step width equal to π and height equal to 1. Similar representations can be given to other event classes; e.g. sporadic events, periodic events with jitters, etc.

Similar to lower and upper *arrival curves* of event streams, a resource r can be described by lower and upper *service curves* r^l and r^u, respectively. If $r[t_1, t_2)$ denotes the number of processing or communication units available from the resource over the time interval $[t_1, t_2)$, then $\forall t_1 < t_2 : r^l(t_2 - t_1) \leq r[t_1, t_2) \leq r^u(t_2 - t_1)$. Service curves of a processor can be represented as lines. More complicated resources (e.g. time division multiplex bus) can be considered in this framework.

Given the arrival curves of an event stream arriving at a resource, and the service curves offered by that resource, it is possible to compute the timing properties of the resulted stream and remaining resource capacity, as well as the maximum backlog and delay experienced by the stream. If $\bar{s}$ is the resulted stream, then we have that

$$\bar{s}^l(t) = \min\{\inf_{0\leq t_1\leq t}\{\sup_{t_2\geq 0}\{s^l(t_1+t_2) - r^u(t_2)\} + r^l(t-t_1)\}, r^l(t)\} \tag{1.18}$$

$$\bar{s}^u(t) = \min\{\sup_{t_2\geq 0}\{\inf_{0\leq t_1\leq t_2+t}\{s^u(t_1) + r^u(t_2+t-t_1)\} - r^l(t_2)\}, r^u(t)\} \tag{1.19}$$

Furthermore, if $\bar{r}$ is the remaining service, then we have that

$$\bar{r}^l(t) = \sup_{0\leq t_1\leq t}\{r^l(t_1) - s^u(t_1)\} \qquad \bar{r}^u(t) = \max\{\inf_{t_1\geq t}\{r^u(t_1) - s^l(t_1)\}, 0\} \tag{1.20}$$

The resulted stream can be processed by another resource and so on. When multiple event enter a single resource, this latter is shared between streams according to a given scheduling policy (e.g. fixed-priority, EDF, TDMA). Thus, arrival curves of the resulted streams depend on what scheduling policy is used.

1.6 Conclusion

Through this section, we have presented a bunch of MOCs that are being used in real-time embedded system design. Three kinds of MOCs are presented: dataflow models, periodic real-time task models, and RTC. Dataflow models offer a simple design framework that guarantees functional determinism. The periodic task model offers theories that ensure temporal predictability. RTC offers a general framework for approximate performance analysis of embedded systems. Through a combination of all these models, this thesis presents a new real-time scheduling approach of dataflow graphs.

Chapter 2

Abstract schedules

Contents

2.1 Priority-driven operational semantics 52

2.2 Activation-related schedules 53

2.2.1 Activation relations 53

2.2.2 Consistency 55

2.2.3 Overflow analysis 63

2.2.4 Underflow analysis 65

2.3 Affine schedules 67

2.3.1 Affine relations 68

2.3.2 Consistency 70

2.3.3 Fixed-priority schedules 72

2.3.4 EDF schedules 76

2.4 Specific cases 78

2.4.1 Ultimately cyclo-static dataflow graphs 79

2.4.2 Multichannels 81

2.4.3 Shared storage space 82

2.4.4 FRStream 83

2.5 Conclusion 86

This chapter describes how to construct abstract priority-driven schedules of dataflow specifications which respect the Kahn principle and hence guarantee functional determinism. An *abstract* schedule consists only of physical-time independent constraints called *activation relations*; and it is constructed in a machine-independent manner. Indeed, we do not consider neither the implementation code of the firing functions nor their worst-case execution times. Therefore, the presented scheduling method requires only an untimed dataflow graph and a real-time scheduling policy.

The optimization problem addressed in this chapter is memory usage minimization. Since each actor is implemented as a separated thread, code size is no longer an issue. Buffering requirements minimization is hence the only optimization problem to consider. Most of existing buffer minimization techniques (mostly in static-periodic scheduling of (C)SDF graphs) are machine-dependent. Though (theoretically speaking) time-dependent techniques are more accurate, the smallest change in execution times (either by changing the target architecture or the implementation code) requires the schedule reconstruction.

We firstly describe what is a consistent and valid abstract schedule by using only a very simple mathematical concept: *sequences*. Then, we show how to construct (using integer linear programming) a specific class of abstract schedules called *affine schedules* with respect to EDF and fixed-priority scheduling policies.

2.1 Priority-driven operational semantics

In a data-driven semantics, an actor is enabled whenever there are enough tokens on its input ports. If channels are bounded, then an actor is enabled only if there are enough empty spaces on its output channels. The scheduler must maintain a list of enabled actors and chooses among them which actors to fire depending on the available resources.

In a clock-triggered semantics, an actor is enabled (or triggered) according to its activation clock. Hence, the list of enabled actors can be constructed much easily in a clock-triggered scheduling compared with a data-driven scheduling. Let $G = (P, E)$ be a static-dataflow graph (see Definition 1.5, p. 26) that consists of N actors. An activation clock is defined as follows.

Definition 2.1 (Activation clock)**.** Let $(\mathcal{T}, \sqsubseteq)$ be a tag system. An activation (release or dispatch) clock of an actor p_i is a total function $\hat{p}_i : C \to \mathbb{N}$ with C a chain in $\mathcal{T}$ such that $\hat{p}_i(t)$ indicates the number of released jobs of actor p_i at tag t.

Example 2.1 (Self-timed example)**.** Consider the self-timed execution in Figure 1.3 (p. 29). The tag system is $(\mathbb{N}, \leq)$; i.e. the system has a global discrete clock. Since the self-timed execution consists of a transient and a periodic phase, activation clocks can be described using ultimately periodic integer sequences. So, $\hat{p}_1 = 0^{30}(2\ 0^{39})^\omega$, $\hat{p}_2 = (1\ 0^{39})^\omega$, and $\hat{p}_3 = 1\ 0^9(3\ 0^{39})^\omega$.

Let t_1 and t_2 be two tags in the chain C such that: (i) at least one job of p_i is released at tag t_1 (i.e. $\hat{p}_i(t_1) > 0$); and (ii) t_2 is the first successor of t_1 at which a job of p_i is released (i.e. $t_2 = \min\{t | t > t_1 \wedge \hat{p}_i(t) > 0\}$). In this thesis, we will assume that "*all jobs of p_i released at tag t_1 complete their execution by tag t_2*". Consequently, we will not consider task sets with deadlines greater than periods.

In a priority-driven operational semantics, each actor p_i is associated with an infinite integer sequence $\omega_i \in \mathbb{N}^\omega$ such that $\omega_i(j)$ is the priority of the jth job of p_i (denoted by $p_i[j]$). We assume that 1 is the highest priority. At each instant t, the scheduler chooses among all the enabled jobs the job with the highest priority. Preemption may occur if

a job with a higher priority than the jobs being executed is enabled. In RM scheduling, each priority sequence is a constant infinite sequence. In EDF scheduling of periodic task sets, priorities of jobs depend on deadlines and phases of actors; priority sequences are hence ultimately periodic. A simple procedure to compute priority sequences in EDF scheduling of periodic task sets (we assume that there is no dynamic admission of new tasks) is to order jobs in a hyperperiod by their deadlines and then assign priority 1 to the job with the earliest deadline, and so on. Priority sequences can model fixed task priority scheduling (e.g. RM), fixed job priority scheduling (e.g. EDF), but not dynamic job priority scheduling. For example, in Least Laxity First (LLF) scheduling, a single job may have different priorities at different moments.

If several jobs of an actor p_i are released at the same tag, then we assume that "*the released jobs read their required data and write their results on a given channel in the same order of activation*"; i.e. job $p_i[j]$ cannot write any token on a given channel before $p_i[j-1]$ writes all its results on that channel; and $p_i[j]$ cannot read any token from a given channel before $p_i[j-1]$ reads all its needed data from that channel. This is a requirement for a functionally deterministic execution of the dataflow graph. In the self-timed example, jobs $p_3[2], p_3[3]$, and $p_3[4]$ execute in parallel with each other. Since the execution time of a job may depend on the data being processed, these three jobs write their results in a non-deterministic order. Recall that if an actor contains some local state, then auto-concurrency must be disabled (by adding self-loops) to ensure proper state update.

In most of dataflow models, actors are assumed to read tokens before performing any computation and then write the results at the end. In our model of computation, we get rid of this constraint in order to give the programmer total freedom on how to write the implementation code of firing functions.

In Kahn semantics of dataflow graphs, an actor blocks if it attempts to read from an empty channel. If channels are bounded, an actor also blocks if it attempts to write on a full channel. In our priority-driven operational semantics, an *overflow exception* occurs when an actor attempts to write to a full channel; while an *underflow exception* occurs when an actor attempts to read from an empty channel.

2.2 Activation-related schedules

An abstract schedule consists of a set of timeless (i.e. abstract) scheduling constraints called activation relations. This section presents the necessary conditions that must be satisfied in order to have a consistent (i.e. no causality problems) and valid (i.e. overflow/underflow-free communications) schedule.

2.2.1 Activation relations

Let $\hat{p}_i, \hat{p}_k : C \to \mathbb{N}$ be two activation clocks defined on the same set of tags. An activation relation is a concise abstract representation of the two clocks on the set $\{t | \hat{p}_i(t) > 0 \vee \hat{p}_k(t) > 0\} \subseteq C$; i.e., instants at which neither actors are released are neglected. This is a time abstraction that keeps only the relative positioning of activations. As

illustrated in Figure 2.1, the duration between releases (i.e. the original timing) does not matter. Hence, only the precedence relations between releases are preserved.

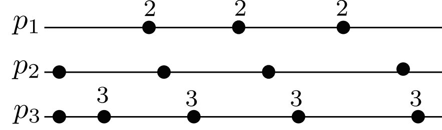


Figure 2.1: Time abstraction of the self-timed example.

Definition 2.2 (Activation relation)**.** Two actors p_i and p_k are said to be activation-related if and only if the relative positioning of all their releases can be entirely defined; that is, $\forall j, j' \in \mathbb{N}_{>0}$: job $p_i[j]$ is released before, after, or at the same time with job $p_k[j']$.

An activation relation can be described by two integer sequences $s_{i,k}$ and $s_{k,i}$ such that $s_{i,k}(j)$ indicates the number of releases of actor p_i at the jth tag in the set $\{t|\hat{p}_i(t) > 0 \vee \hat{p}_k(t) > 0\}$. Hence, $\forall j \in \mathbb{N}_{>0} : s_{i,k}(j) + s_{k,i}(j) > 0$. We say then that p_i and p_k are $[s_{i,k}, s_{k,i}]$-activation-related (denoted by $p_i \overset{[s_{i,k},s_{k,i}]}{\longrightarrow} p_k$); or equivalently p_k and p_i are $[s_{k,i}, s_{i,k}]$-activation-related.

Example 2.2. In the self-timed example, we have that $p_1 \overset{[(0\ 2)^\omega,(1\ 0)^\omega]}{\longrightarrow} p_2 \overset{[(1\ 0)^\omega,1(3\ 0)^\omega]}{\longrightarrow} p_3 \overset{[1(3\ 0)^\omega,0(0\ 2)^\omega]}{\longrightarrow} p_1$.

Since we are interested only in fair and complete executions of live dataflow graphs, we impose the following property.

Property 2.1 (Activation relation)**.** If $[s_{i,k}, s_{k,i}]$ is an activation relation, then

(i) $s_{i,k}, s_{k,i} \in \mathbb{N}^\omega$.

(ii) $\forall j \in \mathbb{N}_{>0} : s_{i,k}(j), s_{k,i}(j)$ and $\oplus s_{i,k}(j) - \oplus s_{k,i}(j)$ are bounded.

Hence, if p_i and p_k are activation-related, then both actors will fire infinitely often (liveness) and between every two releases of an actor there is a finite number of releases of the other actor (fairness). If auto-concurrency is disabled, then we have that $s_{i,k}, s_{k,i} \in \mathbb{B}^\omega$.

Definition 2.3 (Strict predecessors and successors)**.** $\mathsf{sprd}_{i,k} : \mathbb{N}_{>0} \to \mathbb{N}$ is an integer function such that $p_k[\mathsf{sprd}_{i,k}(j)]$ is the last job of p_k which is released *strictly before* $p_i[j]$. Similarly, $\mathsf{ssuc}_{i,k} : \mathbb{N}_{>0} \to \mathbb{N}$ is an integer function such that $p_k[\mathsf{ssuc}_{i,k}(j)]$ is the first job of p_k which is released *strictly after* $p_i[j]$. Both sprd and ssuc are monotone functions.

We have that job $p_i[j]$ is released at the tag number $s_{i,k}^{-1}(j)$ in the set $\{t|\hat{p}_i(t) > 0 \vee \hat{p}_k(t) > 0\}$. All jobs of p_k released before that tag are strict predecessors of $p_i[j]$. Therefore, $\mathsf{sprd}_{i,k}(j) = \oplus s_{k,i}(s_{i,k}^{-1}(j) - 1)$. Similarly, the first strict successor of $p_i[j]$ is the first job of p_k released at a tag greater than $s_{i,k}^{-1}(j)$. Therefore, $\mathsf{ssuc}_{i,k}(j) = \oplus s_{k,i}(s_{i,k}^{-1}(j)) + 1$. By convention, we take $\mathsf{sprd}_{i,k}(0) = 0$ and $\mathsf{ssuc}_{i,k}(0) = 1$.

Definition 2.4 (Predecessors and successors)**.** $\mathsf{prd}_{i,k} : \mathbb{N}_{>0} \to \mathbb{N}$ is an integer function such that $p_k[\mathsf{prd}_{i,k}(j)]$ is the last job of p_k which is released *before or simultaneously* with $p_i[j]$. Similarly, $\mathsf{suc}_{i,k} : \mathbb{N}_{>0} \to \mathbb{N}$ is an integer function such that $p_k[\mathsf{suc}_{i,k}(j)]$ is the first job of p_k which is released *after or simultaneously* with $p_i[j]$. Both prd and suc are monotone functions.

We have that

$$\begin{cases} \mathsf{prd}_{i,k}(j) = \oplus s_{k,i}(s_{i,k}^{-1}(j)) \\ \mathsf{suc}_{i,k}(j) = \oplus s_{k,i}(s_{i,k}^{-1}(j) - 1) + 1 \end{cases}$$

Let $\mathsf{lsim}_{i,k}(j) = \oplus s_{i,k}(s_{i,k}^{-1}(j))$. So, $p_i[\mathsf{lsim}_{i,k}(j)]$ is the last job of p_i which is released simultaneously with $p_i[j]$ according to the activation relation $[s_{i,k}, s_{k,i}]$. Dually, if $\mathsf{fsim}_{i,k}(j) = \oplus s_{i,k}(s_{i,k}^{-1}(j) - 1) + 1$, then $p_i[\mathsf{fsim}_{i,k}(j)]$ is the first job of p_i which is released simultaneously with $p_i[j]$ according to the activation relation $[s_{i,k}, s_{k,i}]$.

Example 2.3. In the self-timed example, we have that $\forall j : \mathsf{sprd}_{1,2}(j) = \left\lceil \frac{j}{2} \right\rceil$, $\mathsf{ssuc}_{1,3}(j) = 3\left\lceil \frac{j}{2} \right\rceil + 2$, and $\mathsf{lsim}_{1,2}(j) = 2\left\lceil \frac{j}{2} \right\rceil$.

Lemma 2.1. $\forall j \in \mathbb{N}_{>0} : \mathsf{prd}_{i,k}(j) < j' \Leftrightarrow \mathsf{prd}_{i,k}(j) < \mathsf{fsim}_{k,i}(j')$ and $\mathsf{suc}_{i,k}(j) > j' \Leftrightarrow \mathsf{suc}_{i,k}(j) > \mathsf{lsim}_{k,i}(j')$.

Definition 2.5 (Synchronous relation)**.** Jobs $p_i[j]$ and $p_k[j']$ are synchronous (i.e. released at the same time) if and only if $\mathsf{sprd}_{i,k}(j) < j' < \mathsf{ssuc}_{i,k}(j)$.

If $p_i[j]$ is released strictly before $p_k[j']$, then we denote that by $p_i[j] < p_k[j']$. If both jobs are synchronous, we denote that by $p_i[j] = p_k[j']$. Knowing functions $\mathsf{sprd}_{i,k}$ and $\mathsf{ssuc}_{i,k}$ is not enough to construct the relative positioning of releases of p_i and p_k unless $s_{i,k}, s_{k,i} \in \mathbb{B}^\omega$ (i.e. auto-concurrency is disabled). In the self-timed example, we have that $\mathsf{sprd}_{2,1}(j) = 2(j-1)$ and $\mathsf{ssuc}_{2,1}(j) = 2j - 1$, but, it is not possible to know only from these two functions that $\forall j \in \mathbb{N}_{>0} : p_1[2j-1] = p_1[2j]$. This means that is possible to deduce $\mathsf{sprd}_{i,k}$ and $\mathsf{ssuc}_{i,k}$ from $[s_{i,k}, s_{k,i}]$ but *not vice versa*.

Property 2.2. The synchronous relation is an equivalence relation; i.e. it is reflexive, symmetric, and transitive.

2.2.2 Consistency

Definition 2.6 (Graph of activation relations)**.** The graph of activation relations $G_r = (P, R)$ is an undirected graph where the set of vertices P represent actors and the set of edges R represent activation relations.

Recall that each activation relation can be reversed. Hence, each edge can be traversed in both directions. By convention, an undirected edge is annotated by an activation relation $[s_{i,k}, s_{k,i}]$ such that $i < k$.

A *walk* in a graph $G = (P, E)$ is an alternating sequence $p_1 \xrightarrow{e_1} p_2 \cdots \xrightarrow{e_m} p_{m+1}$ of vertices and directed edges in P and E respectively so that $\forall i : e_i = (p_i, p_{i+1})$. A walk

is called *closed* if $p_1 = p_{m+1}$. A *path* is a walk with no vertex and no edge repeated. A *cycle* (or simple cycle) is a closed path (i.e. the only repeated vertices are the first and the last ones). For every walk ψ, we denote by ψ^{-1} the inverse walk that starts at the last vertex of ψ and ends at the first vertex.

Let $\psi = p_1 \overset{[s_{1,2},s_{2,1}]}{\longrightarrow} p_2 \longrightarrow \cdots \overset{[s_{m-1,m},s_{m,m-1}]}{\longrightarrow} p_m$ be a walk in the graph of activation relations. Functions sprd_ψ and ssuc_ψ are the extensions of sprd and ssuc using transitivity relation on walk ψ; i.e. $\forall j \in \mathbb{N}_{>0} : \mathsf{sprd}_\psi(j) = (\mathsf{sprd}_{m-1,m} \circ \cdots \circ \mathsf{sprd}_{2,3} \circ \mathsf{sprd}_{1,2})(j)$ and $\mathsf{ssuc}_\psi(j) = (\mathsf{ssuc}_{m-1,m} \circ \cdots \circ \mathsf{ssuc}_{2,3} \circ \mathsf{ssuc}_{1,2})(j)$. If $\Psi_{i,k}$ is the set of all walks from p_i to p_k in G_r, then we put $\mathsf{sprd}_{i\to k}(j) = \max\limits_{\psi\in\Psi_{i,k}} \{\mathsf{sprd}_\psi(j)\}$ and $\mathsf{ssuc}_{i\to k}(j) = \min\limits_{\psi\in\Psi_{i,k}} \{\mathsf{ssuc}_\psi(j)\}$. So, for instance, $p_k[\mathsf{sprd}_{i\to k}(j)]$ is the last job of p_k known from the activation relations that it is released strictly before $p_i[j]$. Similarly, we define functions $\mathsf{prd}_{i\to k}$ and $\mathsf{suc}_{i\to k}$.

Example 2.4. In the self-timed example, we have that $\mathsf{prd}_{1,2}(j) = \left\lceil \frac{j}{2} \right\rceil$, $\mathsf{prd}_{2,3}(j) = 3j-2$, and $\mathsf{prd}_{1,3}(j) = 3\left\lceil \frac{j}{2} \right\rceil + 1$; hence, $\mathsf{prd}_{1,2,3}(j) = 3\left\lceil \frac{j}{2} \right\rceil - 2 < \mathsf{prd}_{1,3}(j)$.

Suppose that we have $P = \{p_1, p_2, p_3\}$ and $R = \{p_1 \overset{[1^\omega,1^\omega]}{\longrightarrow} p_2, p_2 \overset{[1^\omega,1^\omega]}{\longrightarrow} p_3, p_3 \overset{[1^\omega,2^\omega]}{\longrightarrow} p_1\}$. The set R is clearly non-consistent because according to the first and second activation relations $\forall j \in \mathbb{N}_{>0} : p_1[j], p_2[j]$ and $p_3[j]$ are synchronous while $\forall p_i : p_i[j]$ and $p_i[j+1]$ are not synchronous. But according to the third relation, $\forall j \in \mathbb{N}_{>0} : p_3[j]$ and $p_1[2j]$ are synchronous.

Property 2.3. The graph G_r is consistent only if for every actor p_i, if $[s_{i,k}, s_{k,i}]$ and $[s_{i,k'}, s_{k',i}]$ are two activation relations in R, then $\forall j \in \mathbb{N}_{>0} : \mathsf{fsim}_{i,k}(j) = \mathsf{fsim}_{i,k'}(j)$ and $\mathsf{lsim}_{i,k}(j) = \mathsf{lsim}_{i,k'}(j)$.

Property 2.3 means that all activation relations agree on which jobs of a given actor are synchronous with each other. In the sequel, we will suppose that this trivial necessary condition is satisfied. Hence, for sake of conciseness, $\mathsf{fsim}_i(j)$ and $\mathsf{lsim}_i(j)$ denote $\mathsf{fsim}_{i,k}(j)$ and $\mathsf{lsim}_{i,k}(j)$ $(\forall k \neq i)$, respectively. When auto-concurrency is disabled, Property 2.3 is obviously satisfied since $\forall p_i, p_k \in P : \forall j : \mathsf{fsim}_{i,k}(j) = \mathsf{lsim}_{i,k}(j) = j$.

Proposition 2.1 (Consistency of a closed walk)**.** A closed walk $\psi = p_1 \to p_2 \to \cdots \to p_m \to p_1$ in the graph of activation relations is consistent if and only if

$$\forall j \in \mathbb{N}_{>0} : \begin{cases} \mathsf{prd}_\psi(j) = \mathsf{lsim}_1(j) \wedge \mathsf{suc}_\psi(j) = \mathsf{fsim}_1(j) & \textit{if } \psi[j] \textit{ is true} \\ \mathsf{prd}_\psi(j) < \mathsf{fsim}_1(j) \wedge \mathsf{suc}_\psi(j) > \mathsf{lsim}_1(j) & \textit{otherwise} \end{cases} \tag{2.1}$$

where $\psi[j]$ is true if and only if

$$p_1[j] = p_2[\mathsf{suc}_{1,2}(j)] = p_3[\mathsf{suc}_{1,2,3}(j)] = \cdots = p_m[\mathsf{suc}_{1,2,\ldots,m}(j)]$$

Proposition 2.1 ensures that there is no causality problems in the abstract schedule. Causality problems can occur only when there are cycles in the graph of activation

relations. We have that $\mathsf{prd}_\psi(j)$ is the last job of p_1 known (according to the closed walk ψ) to be released before or simultaneously with job $p_1[j]$. So, if $\psi[j]$ is true, then $\mathsf{prd}_\psi(j)$ must be equal to the last job of p_1 released simultaneously with $p_1[j]$ (i.e. $\mathsf{lsim}_1(j)$). If $\psi[j]$ is false, then $\mathsf{prd}_\psi(j)$ must be less than j; but according to Lemma 2.1, we have $\mathsf{prd}_\psi(j) < \mathsf{fsim}_1(j)$. A similar argument holds for $\mathsf{suc}_\psi(j)$.

Example 2.5 (Consistency)**.** Let $P = \{p_1, p_2, p_3\}$ and the closed walk $\psi = p_1 \overset{[3(2\ 1\ 0)^\omega, 0(3\ 0\ 2)^\omega]}{\longrightarrow} p_2 \overset{[0(3\ 0\ 2\ 0)^\omega, 2(1\ 2\ 0\ 1)^\omega]}{\longrightarrow} p_3 \overset{[s_{3,1}, s_{1,3}]}{\longrightarrow} p_1$. The first and second activation relations are depicted in Figure 2.2(a). We use Proposition 2.1 to deduce the possible activation relations between p_3 and p_1 such that ψ is consistent.

Let s' denotes the sequence s where zeros are neglected. To satisfy Property 2.3, we must have $s'_{1,3} = 3(2\ 1)^\omega$ and $s'_{3,1} = 2(1\ 2\ 1)^\omega$. Using Equation 2.1, we have the following constraints.

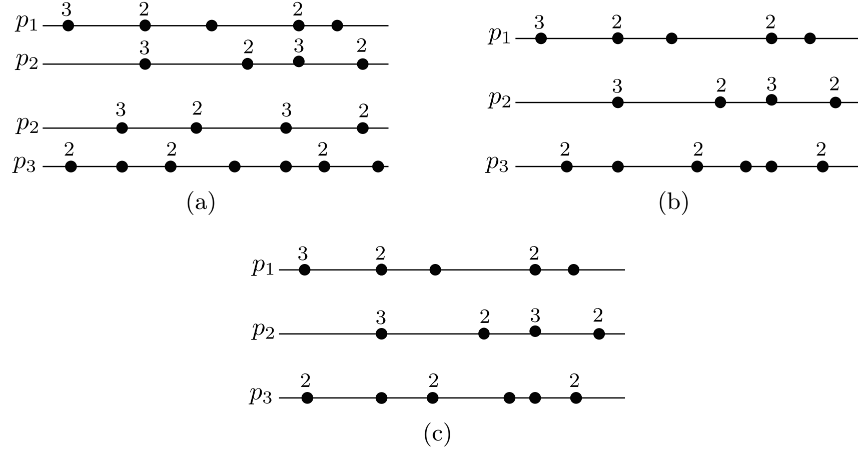


Figure 2.2: Illustration of Proposition 2.1.

$$
\begin{array}{ll}
\mathsf{prd}_\psi(1) = \mathsf{prd}_\psi(2) = \mathsf{prd}_\psi(3) = 0 & \mathsf{suc}_\psi(1) = \mathsf{suc}_\psi(2) = \mathsf{prd}_\psi(3) > 1 \\
\mathsf{prd}_\psi(4) = \mathsf{prd}_\psi(5) = 5 \ \ (\psi[5] \text{ is true}) & \mathsf{suc}_\psi(4) = \mathsf{suc}_\psi(5) = 4 \\
\mathsf{prd}_\psi(6) < 6 & \mathsf{suc}_\psi(6) > 6 \\
\mathsf{prd}_\psi(7) = \mathsf{prd}_\psi(8) = 8 & \mathsf{suc}_\psi(7) = \mathsf{suc}_\psi(8) = 7 \\
\vdots & \vdots
\end{array}
$$

Hence,

$$
\begin{array}{ll}
\mathsf{prd}_{3,1}(3) = 5 & \mathsf{suc}_{3,1}(3) = 4 \\
\mathsf{prd}_{3,1}(7) = 8 & \mathsf{suc}_{3,1}(6) > 6 \\
\mathsf{prd}_{3,1}(11) = 11 & \mathsf{suc}_{3,1}(7) = 7 \\
\mathsf{prd}_{3,1}(15) = 14 & \mathsf{suc}_{3,1}(10) > 9 \\
\vdots & \vdots
\end{array}
$$

There is an infinite number of activation relations that satisfy the previous constraints; for instance, $p_3 \overset{[0\ 2(1\ 0\ 2\ 1)^\omega,3\ 0(2\ 1\ 0\ 0)^\omega]}{\longrightarrow} p_1$ (illustrated in Figure 2.2(b)) and $p_3 \overset{[2(1\ 2\ 1)^\omega,3(2\ 1\ 0)^\omega]}{\longrightarrow} p_1$ (illustrated in Figure 2.2(c)).

Lemma 2.2. Let $\psi_1 = p_1 \to p_2 \to p_3 \to \cdots \to p_1$ and $\psi_2 = p_2 \to p_3 \to \cdots \to p_1 \to p_2$ be two closed walks in G_r. If Equation 2.1 holds (only) for walk ψ_1, then $\forall j \in \mathbb{N}_{>0}$: $\psi_2[\mathsf{prd}_{1,2}(j)]$ is false $\implies \mathsf{prd}_{1,2}(\mathsf{prd}_{\psi_1}(j)) < \mathsf{prd}_{1,2}(j)$.

Proof:
Let us consider all the three possible cases (illustrated in Figure 2.3).

Figure 2.3(a): If $\psi_1[j]$ is true, then $\mathsf{prd}_{\psi_1}(j) = \mathsf{lsim}_1(j)$. Hence, $\mathsf{prd}_{1,2}(\mathsf{prd}_{\psi_1}(j)) = \mathsf{prd}_{1,2}(\mathsf{lsim}_1(j)) = \mathsf{prd}_{1,2}(j)$.

Figure 2.3(b): If $\psi_1[j]$ is false and $\psi_2[\mathsf{prd}_{1,2}(j)]$ is true, then $\mathsf{prd}_{1,2}(\mathsf{prd}_{\psi_1}(j)) = \mathsf{prd}_{1,2}(j)$.

Figures 2.3(c) and 2.3(d): In both cases, $\psi_2[\mathsf{prd}_{1,2}(j)]$ is false. Suppose that $\mathsf{prd}_{1,2}(\mathsf{prd}_{\psi_1}(j)) \geq \mathsf{prd}_{1,2}(j)$. Since $\psi_1[j]$ is false, we have that $\mathsf{prd}_{\psi_1}(j) < \mathsf{fsim}_1(j)$; and hence $\mathsf{prd}_{1,2}(\mathsf{prd}_{\psi_1}(j)) = \mathsf{prd}_{1,2}(j)$. This implies that $\mathsf{prd}_{\psi_1}(\mathsf{prd}_{\psi_1}(j)) = \mathsf{prd}_{\psi_1}(j)$; hence, $\psi_1[\mathsf{prd}_{\psi_1}(j)]$ is true. But $\psi_1[\mathsf{prd}_{\psi_1}(j)]$ cannot be true, otherwise $\psi_2[\mathsf{prd}_{1,2}(j)]$ will also be true. Contradiction. □

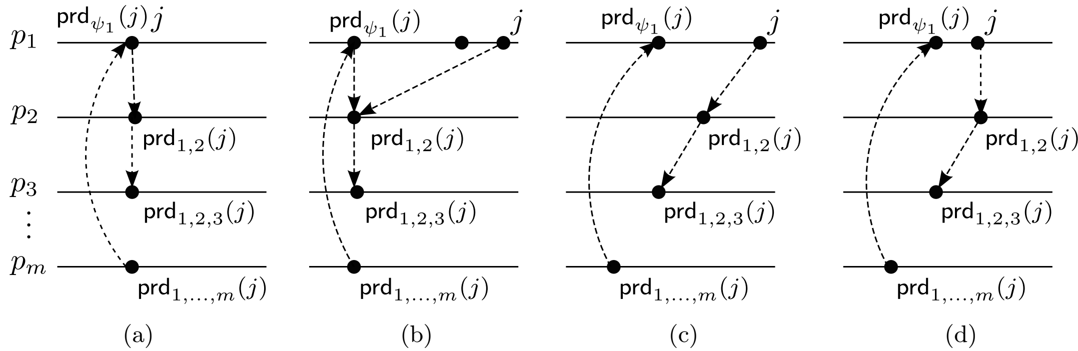


Figure 2.3: Illustration of Lemma 2.2.

In a similar way, we can prove that if $\psi_2[\mathsf{suc}_{1,2}(j)]$ is false, then $\mathsf{suc}_{1,2}(\mathsf{suc}_{\psi_1}(j)) > \mathsf{suc}_{1,2}(j)$.

Proposition 2.2. If Equation 2.1 holds for the closed walk $\psi_1 = p_1 \to p_2 \to \cdots \to p_i \to p_{i+1} \to \cdots \to p_m \to p_1$, then it holds for any closed walk $\psi_i = p_i \to p_{i+1} \to \cdots \to p_{i-1} \to p_i$ for $i = 1, m$.

Proof:
We prove that if Equation 2.1 holds for ψ_1, then it holds for ψ_2; and so it holds for ψ_3 and so on. We first have to prove the first part; i.e.

$$\forall j' \in \mathbb{N}_{>0} : \begin{cases} \mathsf{prd}_{\psi_2}(j') = \mathsf{lsim}_2(j') & \textit{if } \psi_2[j'] \textit{ is true} \\ \mathsf{prd}_{\psi_2}(j') < \mathsf{fsim}_2(j') & \textit{otherwise} \end{cases}$$

We have that $\mathsf{prd}_{\psi_2}(j') = \mathsf{prd}_{1,2} \circ \mathsf{prd}_{m,1} \circ \cdots \circ \mathsf{prd}_{2,3}(j')$. We consider two cases:

1) If $\exists j \in \mathbb{N}_{>0} : j' \in [\mathsf{fsim}_2(\mathsf{prd}_{1,2}(j)), \mathsf{prd}_{1,2}(j)]$, then $\mathsf{prd}_{2,3}(j') = \mathsf{prd}_{2,3} \circ \mathsf{prd}_{1,2}(j)$ and hence $\mathsf{prd}_{\psi_2}(j') = \mathsf{prd}_{1,2}(\mathsf{prd}_{\psi_1}(j))$. As in the proof of Lemma 2.2, we have three possibilities.

- $\psi_1[j]$ is true: In this case, $\psi_2[j']$ is also true. So, $\mathsf{prd}_{\psi_2}(j') = \mathsf{prd}_{1,2}(\mathsf{prd}_{\psi_1}(j)) = \mathsf{prd}_{1,2}(\mathsf{lsim}_1(j)) = \mathsf{lsim}_2(j')$.
- $\psi_1[j]$ is false and $\psi_2[j']$ is true: In this case, $\mathsf{prd}_{1,2}(\mathsf{prd}_{\psi_1}(j)) = \mathsf{prd}_{1,2}(j)$. Hence, $\mathsf{prd}_{\psi_2}(j') = \mathsf{prd}_{1,2}(j) = \mathsf{lsim}_2(j')$.
- $\psi_2[j']$ is false: According to Lemma 2.2, we have that $\mathsf{prd}_{1,2}(\mathsf{prd}_{\psi_1}(j)) < \mathsf{prd}_{1,2}(j)$. Hence, $\mathsf{prd}_{\psi_2}(j') < \mathsf{prd}_{1,2}(j)$. So, according to Lemma 2.1, $\mathsf{prd}_{\psi_2}(j') < \mathsf{fsim}_2(\mathsf{prd}_{1,2}(j)) = \mathsf{fsim}_2(j')$.

2) If $\nexists j \in \mathbb{N}_{>0} : j' \in [\mathsf{fsim}_2(\mathsf{prd}_{1,2}(j)), \mathsf{prd}_{1,2}(j)]$, then $\psi_2[j']$ cannot be true. Let us take j such that $j = \mathsf{suc}_{2,1}(j')$. There are two possibilities, as illustrated in Figure 2.4.

- $\psi_1[j]$ is true: In this case, $\mathsf{prd}_{2,\ldots,1}(j') < j$. Therefore, $\mathsf{prd}_{\psi_2}(j') = \mathsf{prd}_{1,2}(\mathsf{prd}_{2,\ldots,1}(j')) < \mathsf{fsim}_2(j')$ (otherwise $j \neq \mathsf{suc}_{2,1}(j')$).
- $\psi_1[j]$ is false: In this case, $\mathsf{prd}_{\psi_1}(j) < \mathsf{fsim}_1(j)$. Suppose that $\mathsf{prd}_{1,2}(\mathsf{prd}_{\psi_1}(j)) \geq \mathsf{fsim}_2(j')$. As illustrated in Figure 2.4(b), $\mathsf{suc}_{2,1}(j') = \mathsf{prd}_{\psi_1}(j) < \mathsf{fsim}_1(j)$. This contradicts the hypothesis that $j = \mathsf{suc}_{2,1}(j')$. Hence, $\mathsf{prd}_{1,2}(\mathsf{prd}_{\psi_1}(j)) < \mathsf{fsim}_2(j')$. Since, $\mathsf{prd}_{2,\ldots,1}(j') \leq \mathsf{prd}_{\psi_1}(j)$ (due to monotonicity of prd), we have that $\mathsf{prd}_{\psi_2}(j') = \mathsf{prd}_{1,2}(\mathsf{prd}_{2,\ldots,1}(j')) < \mathsf{fsim}_2(j')$.

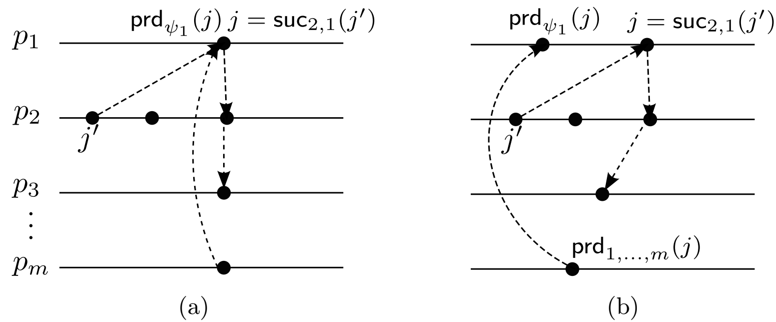


Figure 2.4: Illustration of Proposition 2.2.

Similarly, we can prove that

$$\forall j' \in \mathbb{N}_{>0} : \begin{cases} \mathsf{suc}_{\psi}(j') = \mathsf{fsim}_2(j') & \textit{if } \psi[j'] \textit{ is true} \\ \mathsf{suc}_{\psi}(j') > \mathsf{lsim}_2(j') & \textit{otherwise} \end{cases}$$

□

Proposition 2.3. If Equation 2.1 holds for the closed walk $\psi = p_1 \to p_2 \to \cdots \to p_m \to p_1$, then it holds for walk $\psi' = p_m \to p_{m-1} \to \cdots \to p_1 \to p_m$.

Proof: Firstly, we prove that

$$\forall j' \in \mathbb{N}_{>0} : \begin{cases} \mathsf{prd}_{\psi'}(j') = \mathsf{lsim}_m(j') & \textit{if } \psi'[j'] \textit{ is true} \\ \mathsf{prd}_{\psi'}(j') < \mathsf{fsim}_m(j') & \textit{otherwise} \end{cases}$$

There are two cases:

1) $\psi'[j']$ is true: As illustrated in Figure 2.5(a), if $j = \mathsf{prd}_{m,m-1,\ldots,1}(j')$, then $\psi[j]$ is also true. Hence, $\mathsf{prd}_{\psi'}(j') = \mathsf{prd}_{1,m}(j) = \mathsf{lsim}_m(j')$.

2) $\psi'[j']$ is false: Let $j = \mathsf{prd}_{m,m-1,\ldots,1}(j')$. we have that $\mathsf{suc}_{1,\ldots,m}(j) \leq \mathsf{fsim}_m(j')$ and hence $\mathsf{suc}_{\psi}(j) \leq \mathsf{suc}_{m,1}(j)$. Suppose that $\mathsf{prd}_{\psi'}(j') \geq \mathsf{fsim}_m(j')$ (i.e. the walk ψ' is inconsistent). Hence, $\mathsf{prd}_{1,m}(j) \geq \mathsf{fsim}_m(j')$. This implies that $\mathsf{suc}_{m,1}(j') \leq \mathsf{fsim}_1(j)$. So, we have that $\mathsf{suc}_{\psi}(j) \leq \mathsf{suc}_{m,1}(j') \leq \mathsf{fsim}_1(j) \leq \mathsf{lsim}_1(j)$. This contradicts the hypothesis that Equation 2.1 holds for walk ψ.

Similarly, we can prove that

$$\forall j' \in \mathbb{N}_{>0} : \begin{cases} \mathsf{suc}_{\psi'}(j') = \mathsf{fsim}_m(j') & \textit{if } \psi'[j'] \text{ is true} \\ \mathsf{suc}_{\psi'}(j') > \mathsf{lsim}_m(j') & \textit{otherwise} \end{cases}$$

□

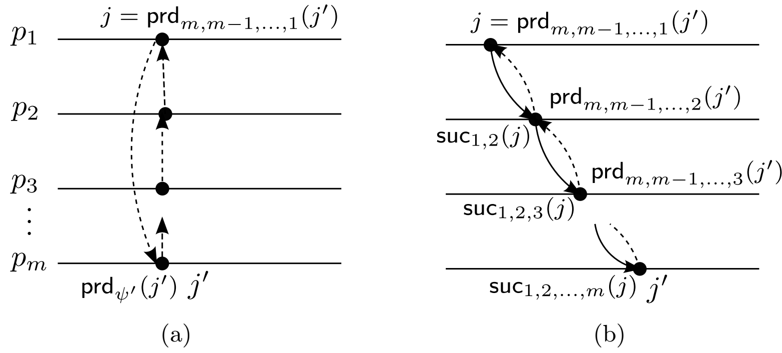


Figure 2.5: Illustration of Proposition 2.3.

Composition

A *cycle* (or simple cycle) is a closed path; i.e. a closed walk with no repeated vertices (except the requisite repetition of the first and last vertices). For example, in Figure 2.6, walks $\psi = p_1 \to p_2 \to p_1$ and $\psi' = p_2 \to p_3 \to p_4 \to p_5 \to p_2$ are simple cycles; while walk $\psi'' = p_1 \to p_2 \to p_3 \to p_4 \to p_5 \to p_2 \to p_1$ is not a simple one. Every closed walk can be expressed as the disjoint union of some simple cycles. For instance, ψ'' is the disjoint union of ψ and ψ'.

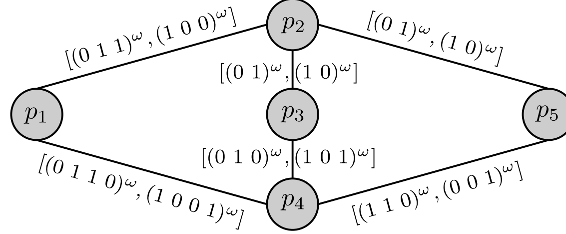


Figure 2.6: Graph of activation relations.

Proposition 2.4 (Consistency of closed walks)**.** A closed walk ψ is consistent if all simple cycles composing ψ are consistent.

Proof:

Let $\psi = p_1 \to \ldots \to p_1$. Suppose that p_i appears more than once in ψ. Hence, ψ can be decomposed as $p_1 \to \ldots \to p_{i-1} \to \psi' \to p_{i+1} \to \ldots \to p_1$ such that ψ' is the walk from the first occurrence of p_i to the last one. Let us put $\psi'' = p_1 \to \ldots \to p_i \to \ldots \to p_1$ (i.e. ψ'' is obtained by removing the sub-walk ψ' from ψ). We suppose that ψ' and ψ'' are simple cycles, otherwise we have to further decompose them. We have that $\forall j \in \mathbb{N}_{>0} : \mathsf{prd}_\psi(j) = \mathsf{prd}_{i,i+1,\ldots,1}(\mathsf{prd}_{\psi'}(\mathsf{prd}_{1,\ldots,i-1,i}(j)))$.

- $\psi[j]$ is true: trivially, $\psi'[\mathsf{prd}_{1,\ldots,i-1,i}(j)]$ and $\psi''[j]$ are true. So, $\mathsf{prd}_{\psi'}(\mathsf{prd}_{1,\ldots,i-1,i}(j)) = \mathsf{prd}_{1,\ldots,i-1,i}(j)$. Hence, $\mathsf{prd}_\psi(j) = \mathsf{prd}_{i,i+1,\ldots,1}(\mathsf{prd}_{1,\ldots,i-1,i}(j)) = \mathsf{prd}_{\psi''}(j) = \mathsf{lsim}_1(j)$.
- $\psi[j]$ is false: In this case, $\psi''[j]$ and $\psi'[\mathsf{prd}_{1,\ldots,i-1,i}(j)]$ cannot be both true. In case $\psi'[\mathsf{prd}_{1,\ldots,i-1,i}(j)]$ is true, we have that $\mathsf{prd}_\psi(j) = \mathsf{prd}_{i,i+1,\ldots,1}(\mathsf{prd}_{\psi'}(\mathsf{prd}_{1,\ldots,i-1,i}(j))) = \mathsf{prd}_{i,i+1,\ldots,1}(\mathsf{prd}_{1,\ldots,i-1,i}(j)) = \mathsf{prd}_{\psi''}(j) < \mathsf{fsim}_1(j)$. In case $\psi''[j]$ is true, we have that $\mathsf{prd}_\psi(j) = \mathsf{prd}_{i,i+1,\ldots,1}(\mathsf{prd}_{\psi'}(\mathsf{prd}_{1,\ldots,i-1,i}(j))) < \mathsf{prd}_{i,i+1,\ldots,1}(\mathsf{fsim}_i$ $(\mathsf{prd}_{1,\ldots,i-1,i}(j))) = \mathsf{lsim}_1(j)$. The third case (i.e. when both $\psi'[\mathsf{prd}_{1,\ldots,i-1,i}(j)]$ and $\psi''[j]$ are false) is straightforward.

Similarly, we can prove the second part of Equation 2.1. □

Some simple cycles can also be expressed as the symmetric difference of some other simple cycles. For instance, in Figure 2.6, we have that $p_1 \to p_2 \to p_3 \to p_4 \to p_1 \uplus p_2 \to p_5 \to p_4 \to p_3 \to p_2 = p_1 \to p_2 \to p_5 \to p_4 \to p_1$.

Question 1 Let ψ, ψ', ψ'' be three simple cycles such that $\psi = \psi' \uplus \psi''$. If ψ' and ψ'' are consistent, does this imply that ψ is also consistent?

Example 2.6 (Counterexample-Q1)**.** In the general case, the answer to question 1 is ***no***. The graph in Figure 2.6 is a counterexample where $\psi' = p_1 \xrightarrow{[(0\ 1\ 1)^\omega,(1\ 0\ 0)^\omega]} p_2 \xrightarrow{[(0\ 1)^\omega,(1\ 0)^\omega]} p_3 \xrightarrow{[(0\ 1\ 0)^\omega,(1\ 0\ 1)^\omega]} p_4 \xrightarrow{[(1\ 0\ 0\ 1)^\omega,(0\ 1\ 1\ 0)^\omega]} p_1$ and $\psi'' = p_2 \xrightarrow{[(0\ 1)^\omega,(1\ 0)^\omega]} p_5 \xrightarrow{[(0\ 0\ 1)^\omega,(1\ 1\ 0)^\omega]} p_4 \xrightarrow{[(1\ 0\ 1)^\omega,(0\ 1\ 0)^\omega]} p_3 \xrightarrow{[(1\ 0)^\omega,(0\ 1)^\omega]} p_2$. Equation 2.1 holds for ψ' (Figure 2.7(a)) and ψ'' (Figure 2.7(b)) but not for $\psi = \psi' \uplus \psi''$ (Figure 2.7(c)). Indeed, $\mathsf{prd}_\psi(1) = 2$.

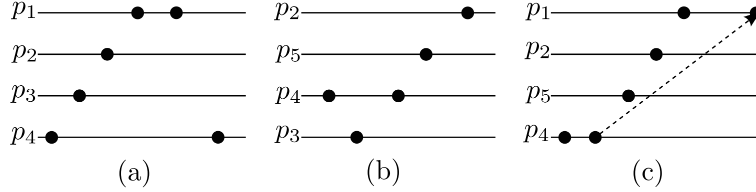


Figure 2.7: Consistency of symmetric difference of simple cycles.

Question 2 Let ψ, ψ', ψ'' be three simple cycles such that $\psi = \psi' \uplus \psi''$. If ψ is consistent, does this imply that ψ' and ψ'' are also consistent?

Example 2.7 (Counterexample-Q2)**.** In the general case, the answer to question 2 is ***no***. In Example 2.6, if we put $p_4 \overset{[(1\ 0\ 1)^\omega,(0\ 1\ 0)^\omega]}{\longrightarrow} p_5$ and $p_3 \overset{[(0\ 0\ 1)^\omega,(1\ 1\ 0)^\omega]}{\longrightarrow} p_4$, then Equation 2.1 holds for cycle ψ but not for ψ'.

Proposition 2.5 (Consistency of an abstract schedule)**.** An abstract schedule is consistent if and only if *every undirected cycle* in the graph of activation relations is consistent.

Proof:
Let $\Psi_{i,i}$ be the set of all closed walks from p_i to itself in G_r. If Equation 2.1 holds for each walk in $\Psi_{i,i}$, then we will have that

$$\forall j \in \mathbb{N}_{>0} : \begin{cases} \mathsf{prd}_{i\to i}(j) = \mathsf{lsim}_i(j) \wedge \mathsf{suc}_{i\to i}(j) = \mathsf{fsim}_i(j) & \textit{if } \exists \psi \in \Psi_{i,i} : \psi[j] \textit{ is true} \\ \mathsf{prd}_{i\to i}(j) < \mathsf{fsim}_i(j) \wedge \mathsf{suc}_{i\to i}(j) > \mathsf{lsim}_i(j) & \textit{otherwise} \end{cases}$$

Hence, there will be no causality problems in the schedule. According to Proposition 2.4 and the previous counterexamples, it is enough to check only the consistency of *all* the simple cycles. According to propositions 2.2 and 2.3, it is enough to check only the consistency of one cycle per undirected cycle in graph G_r. □

Consistency of ultimately periodic schedules

Definition 2.7 (Ultimately periodic schedules)**.** An activation relation $[s_{i,k}, s_{k,i}]$ is ultimately periodic if $s_{i,k}$ and $s_{k,i}$ are ultimately periodic sequences. An abstract schedule is ultimately periodic if all its activation relations are ultimately periodic.

Hence, we can put $s_{i,k} = u_{i,k}v_{i,k}^\omega$ and $s_{k,i} = u_{k,i}v_{k,i}^\omega$ such that $|u_{i,k}| = |u_{k,i}|$ and $|v_{i,k}| = |v_{k,i}|$. Example 2.2 is an ultimately periodic abstract schedule.

Proposition 2.6. A closed walk $\psi = p_1 \to p_2 \to \cdots \to p_m \to p_1$ in an ultimately periodic schedule is consistent *only if*

$$\prod_{i=1}^{m} \|v_{i,k}\| = \prod_{i=1}^{m} \|v_{k,i}\| \tag{2.2}$$

Where $k = i \mod m + 1$ and $\|v\|$ is the sum of elements of the finite integer sequence v.

Proof:

If Equation 2.1 holds for ψ, then we have that $\forall j \in \mathbb{N}_{>0} : \mathsf{prd}_\psi(j) \leq j$ and $\mathsf{suc}_\psi(j) \geq j$. Let k_1 be the maximum integer for which $\|u_{1,2}\| + k_1\|v_{1,2}\| < \mathsf{fsim}_1(j)$. As illustrated in Figure 2.8, k_1 is the number of instances of the positioning pattern before the instance that contains the jth activation of p_1. Since $[s_{1,2}, s_{2,1}]$ is an ultimately periodic activation relation, we have that $\mathsf{prd}_{1,2}(j) \geq j_1 = \|u_{2,1}\| + k_1\|v_{2,1}\|$. Continuing the same process, k_2 is the maximum integer for which $\|u_{2,3}\| + k_2\|v_{2,3}\| < \mathsf{fsim}_2(j_1)$ and so $\mathsf{prd}_{2,3}(j_1) \geq j_2 = \|u_{3,2}\| + k_2\|v_{3,2}\|$. Hence, $j_m = \|u_{1,m}\| + k_m\|v_{1,m}\| \leq \mathsf{prd}_{m,1}(j_{m-1}) \leq j$.

We denote by c any non important constant. We have that $k_1 = \frac{1}{\|v_{1,2}\|}j + c$, $k_2 = \frac{\|v_{2,1}\|}{\|v_{1,2}\|\|v_{2,3}\|}j + c, \ldots, k_m = \frac{\|v_{2,1}\|\ldots\|v_{m,m-1}\|}{\|v_{1,2}\|\ldots\|v_{m,1}\|}j + c$. Let us denote by $\prod_\psi$ the product of $\|v_{i,k}\|$ of all the activation relations on a walk ψ. So, $\mathsf{prd}_\psi(j) \leq j$ implies that $\frac{\prod_{\psi^{-1}}}{\prod_\psi}j + c \leq j$. When j tends to infinity, the previous inequality has a solution only if $\prod_{\psi^{-1}} \leq \prod_\psi$. Similarly and using the second part of Equation 2.1, we have that $\frac{\prod_{\psi^{-1}}}{\prod_\psi}j + c \geq j$. Thus, $\prod_{\psi^{-1}} \geq \prod_\psi$. □

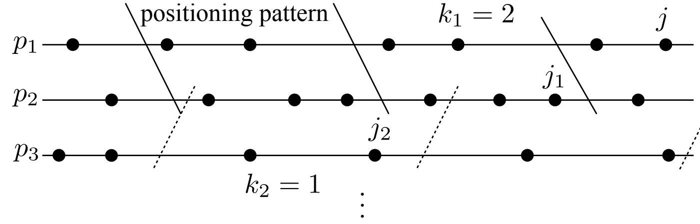


Figure 2.8: Illustration of Proposition 2.6.

Property 2.4. Let ψ, ψ', and ψ'' be three simple cycles such that $\psi = \psi' \uplus \psi''$. If Equation 2.8 holds for ψ' and ψ'', then it holds for ψ.

Proof:

We prove the case where the set of edges in ψ' that have opposites in ψ'' forms a contiguous path; however, the presented argument also holds for the general case. Let $\psi_* = \psi' \cap \psi''$ be such a contiguous path. Hence, we can put $\psi' = \psi_1 + \psi_*$, $\psi'' = \psi_2 + \psi_*^{-1}$, and $\psi = \psi_1 + \psi_2$. Since Equation 2.8 holds for ψ' and ψ'', we have that $\prod_{\psi_1} \cdot \prod_{\psi_*} = \prod_{\psi_1^{-1}} \cdot \prod_{\psi_*^{-1}}$ and $\prod_{\psi_2} \cdot \prod_{\psi_*^{-1}} = \prod_{\psi_2^{-1}} \cdot \prod_{\psi_*}$. Hence, $\prod_{\psi_1} \cdot \prod_{\psi_2} = \prod_{\psi_1^{-1}} \cdot \prod_{\psi_2^{-1}}$. □

2.2.3 Overflow analysis

An overflow exception occurs when an actor attempts to write to a full channel. Let $e = (p_i, p_k, x, y)$ be a channel between the producer p_i and the consumer p_k. The overflow analysis ensures that the (over-approximated) number of accumulated tokens in channel e does not exceed its size $\delta(e)$ for every execution of the dataflow graph. The (over-approximated) number of accumulated tokens can be computed as follows.

- The number of produced tokens by job $p_i[j]$ is equal to $x(j)$.

• The number of produced tokens before $p_i[j]$ writes any token is equal to $\oplus x(j-1)$ since we assume that the scheduling policy ensures, even with auto-concurrency, that jobs of the producer write tokens in the same order of their activation. The number of produced tokens until and including the jth job of p_i is hence equal to $\oplus x(j)$. If $j = \mathsf{lsim}_i(j)$ (which is always the case if auto-concurrency is disabled), then $\oplus x(j) = \oplus(x \otimes s_{i,k})(s_{i,k}^{-1}(j)) = \oplus x(\oplus s_{i,k}(s_{i,k}^{-1}(j)))$.

Definition 2.8 (Complete before)**.** $\mathsf{cbef}_{i,k} : \mathbb{N}_{>0} \to \mathbb{N}$ is an integer function such that $p_k[\mathsf{cbef}_{i,k}(j)]$ is the last job of p_k which *certainly* reads all its needed tokens from channel e *before* $p_i[j]$ writes any token on that channel. So, $\mathsf{cbef}_{i,k}$ is a monotone function.

• The (under-approximated) number of consumed tokens before $p_i[j]$ writes any token on channel e is equal to $\oplus y(\mathsf{cbef}_{i,k}(j))$ since we assume that the scheduling policy ensures, even with auto-concurrency, that jobs of the consumer consume tokens in the same order of their activation. If $\mathsf{cbef}_{i,k}(j) = \mathsf{lsim}_k(\mathsf{cbef}_{i,k}(j))$, then $\oplus y(\mathsf{cbef}_{i,k}(j)) = \oplus(y \otimes s_{k,i})(s_{k,i}^{-1}(\mathsf{cbef}_{i,k}(j))) = \oplus y(\oplus s_{k,i}(s_{k,i}^{-1}(\mathsf{cbef}_{i,k}(j))))$.

Function $\mathsf{cbef}_{i,k}$ depends essentially on the scheduling policy, the physical timing parameters, and the implementation code of the firing functions. Since the abstract schedule consists of a set of timeless constraints, function $\mathsf{cbef}_{i,k}$ could be only a safe approximation that does not consider neither the timing nor the implementation code.

Example 2.8. Let us take $e_2 = (p_1, p_2, 1^\omega, 2^\omega)$ of the self-timed example (Figure 1.1, p. 26). We have that $\mathsf{cbef}_{1,2}(j) = \mathsf{sprd}_{1,2}(j) = \left\lceil \frac{j}{2} \right\rceil$. Since actor p_2 does not preempt actor p_1 or execute in parallel with it, function $\mathsf{cbef}_{1,2}$, as defined before, is accurate. Similarly, $\forall j \geq 2 : \mathsf{cbef}_{2,3}(j) = \mathsf{sprd}_{2,3}(j)$ gives the accurate values. However, since job $p_2[1]$ executes in parallel with job $p_3[1]$, it is not safe to say that $p_3[1]$ reads the required data before $p_2[1]$ writes its results unless we assume that an actor reads tokens at the beginning of firings and writes results at the end. A safe approximation for the overflow analysis is one that over-approximates the number of accumulated tokens in channels, hence $\mathsf{cbef}_{2,3}(1) = 0$.

The (over-approximated) number of accumulated tokens on channel e when job $p_i[j]$ writes all its results can be given as

$$X(j) = \theta(e) + \oplus x(j) - \oplus y(\mathsf{cbef}_{i,k}(j))$$

If the over-approximated number of accumulated tokens in the channel does not exceed the size of the channel $\delta(e)$ for all $j \in \mathbb{N}_{>0}$, then overflow exceptions will not occur at run-time. Hence, the overflow analysis can be formulated as follows.

$$\forall j \in \mathbb{N}_{>0} : \theta(e) + \oplus x(j) - \oplus y(\mathsf{cbef}_{i,k}(j)) \leq \delta(e) \tag{2.3}$$

If $\forall j \in \mathbb{N}_{>0} : \mathsf{cbef}_{i,k}(j) = \mathsf{cbef}_{i,k}(\mathsf{lsim}_i(j))$, then Equation 2.3 can be checked only for points where $j = \mathsf{lsim}_i(j)$ since the maximum numbers of accumulated tokens happen at these points.

Example 2.9. In the self-timed example and for $e = (p_1, p_2, 1^\omega, 2^\omega)$, we have that $s_{2,1} \in \mathbb{B}^\omega$ and $\mathsf{cbef}_{1,2}(j) = \mathsf{cbef}_{1,2}(\mathsf{lsim}_1(j))$. Thus, Equation 2.3 can be written as

$$\forall j \in \mathbb{N}_{>0} : \theta(e) + \oplus x(\mathsf{lsim}_1(j)) - \oplus y(\mathsf{cbef}_{1,2}(j)) \leq \delta(e)$$

We have that $\forall j \in \mathbb{N}_{>0}$:

$$\oplus x(j) = j; \quad \oplus y(j) = 2j; \quad \mathsf{cbef}_{1,2}(j) = \left\lceil \tfrac{j}{2} \right\rceil; \quad \mathsf{lsim}_1(j) = 2\left\lceil \tfrac{j}{2} \right\rceil$$

So,

$$X(j) = \theta(e) + 2\left\lceil \frac{j}{2} \right\rceil - 2\left\lceil \frac{j}{2} \right\rceil = \theta(e)$$

For $\theta(e) = 2$, the overflow equation can be written as

$$\forall j \in \mathbb{N}_{>0} : \delta(e) \geq 2$$

The minimum safe size of channel e is hence equal to 2 (which is equal to the size obtained by the symbolic execution).

Boundedness

Channel e can be implemented as a bounded buffer if there are bounded constants $\theta(e)$ and $\delta(e)$ (with $\theta(e) \leq \delta(e)$) that satisfy the overflow equation. From Equation 2.3, we have that $\oplus y(\mathsf{cbef}_{i,k}(j)) \geq \oplus x(j) + \theta(e) - \delta(e)$. Since $\oplus y$ is a monotone function, we have that

$$\mathsf{cbef}_{i,k}(j) \geq y^{-1}(\oplus x(j) + \theta(e) - \delta(e)) \tag{2.4}$$

According to the second property of an activation relation (Property 2.1), there must be a finite number of firings of the consumer between every two firings of the producer. Hence, function $\mathsf{cbef}_{i,k}$ should satisfy

$$\forall j \in \mathbb{N}_{>0} : \mathsf{cbef}_{i,k}(j) - \mathsf{cbef}_{i,k}(j-1) \text{ is bounded} \tag{2.5}$$

Example 2.10 (SDF graphs)**.** Let $e = (p_1, p_2, a^\omega, b^\omega)$ be a channel in a SDF graph where $a, b \in \mathbb{N}_{>0}$. We have that $\oplus x(j) = aj$, $\oplus y(j) = bj$, and $y^{-1}(j) = \left\lceil \frac{j}{b} \right\rceil$. So, $\mathsf{cbef}_{1,2}(j) \geq \left\lceil \frac{a}{b} j + \frac{\theta(e)-\delta(e)}{b} \right\rceil$. For instance, if the activation relation and the scheduling policy ensure that $\mathsf{cbef}_{1,2}(j) = \max\{0, \left\lceil \frac{a}{b} j + \frac{\theta(e)-\delta(e)}{b} \right\rceil\}$, then $\mathsf{cbef}_{i,k}(j) - \mathsf{cbef}_{i,k}(j-1)$ is upper bounded by $\left\lceil \frac{a}{b} \right\rceil$.

2.2.4 Underflow analysis

The underflow analysis is dual to the overflow analysis. An underflow exception occurs when an actor attempts to read from en empty channel. The underflow analysis ensures that the (under-approximated) number of accumulated tokens is always greater than

or equal to zero for every execution of the dataflow graph. The (under-approximated) number of accumulated tokens can be computed as follows.

• The number of consumed tokens by job $p_k[j]$ is equal to $y(j)$.

• The number of consumed tokens before $p_k[j]$ reads any token is equal to $\oplus y(j-1)$. The number of consumed tokens until and including the jth job of p_k is hence equal to $\oplus y(j)$. If $j = \mathsf{lsim}_k(j)$, then $\oplus y(j) = \oplus(y \otimes s_{k,i}(s_{k,i}^{-1}(j)) = \oplus y(\oplus s_{k,i}(s_{k,i}^{-1}(j)))$.

• The (under-approximated) number of produced tokens before $p_k[j]$ reads any token on channel e is equal to $\oplus x(\mathsf{cbef}_{k,i}(j))$. Function $\mathsf{cbef}_{k,i} : \mathbb{N}_{>0} \to \mathbb{N}$ is a monotone integer function such that $p_i[\mathsf{cbef}_{k,i}(j)]$ is the last job of p_i which ***certainly*** writes all its results on channel e ***before*** $p_k[j]$ reads any token from that channel. If $\mathsf{cbef}_{k,i}(j) = \mathsf{lsim}_i(\mathsf{cbef}_{k,i}(j))$, then $\oplus x(\mathsf{cbef}_{k,i}(j)) = \oplus(x \otimes s_{i,k})(s_{i,k}^{-1}(\mathsf{cbef}_{k,i}(j))) = \oplus x(\oplus s_{i,k}(s_{i,k}^{-1}(\mathsf{cbef}_{k,i}(j))))$.

If the under-approximated number of accumulated tokens in the channel is greater than or equal to zero for all $j \in \mathbb{N}_{>0}$, then underflow exceptions will not occur at run time. Hence, the underflow analysis can be formulated as follows.

$$\forall j \in \mathbb{N}_{>0} : \theta(e) + \oplus x(\mathsf{cbef}_{k,i}(j)) - \oplus y(j) \geq 0 \tag{2.6}$$

Example 2.11. Let us take channel $e = (p_2, p_3, 3^\omega, 1^\omega)$ of the self-timed example. We have that $\forall j \geq 2 : \mathsf{cbef}_{3,2}(j) = \mathsf{sprd}_{3,2}(j) = \left\lceil \frac{j-1}{3} \right\rceil$. Since it is not safe to say that $p_2[1]$ writes its results before $p_3[1]$ reads any token, we must take $\mathsf{cbef}_{3,2}(1) = 0$. We have that $s_{2,3} \in \mathbb{B}^\omega$ and $\mathsf{cbef}_{3,2}(j) = \mathsf{cbef}_{3,2}(\mathsf{lsim}_3(j))$. Thus, Equation 2.6 can be written as

$$\forall j \in \mathbb{N}_{>0} : Y(j) = \theta(e) + \oplus x(\mathsf{cbef}_{3,2}(j)) - \oplus y(\mathsf{lsim}_3(j)) \geq 0$$

We have that

$$\oplus x(j) = 3j; \quad \oplus y(j) = j; \quad \mathsf{lsim}_3(j) = 3\left\lceil \tfrac{j-1}{3} \right\rceil + 1; \quad \mathsf{cbef}_{3,2}(j) = \left\lceil \tfrac{j-1}{3} \right\rceil$$

So,

$$Y(j) = \theta(e) + 3\left\lceil \frac{j-1}{3} \right\rceil - (3\left\lceil \frac{j-1}{3} \right\rceil + 1) = \theta(e) - 1$$

The underflow equation can be written as

$$\forall j \in \mathbb{N}_{>0} : \theta(e) - 1 \geq 0$$

The minimum number of initial tokens on the channel that excludes underflow exceptions is hence equal to 1.

Boundedness

Channel e can be implemented as a bounded buffer if there is a bounded constant $\theta(e)$ that satisfies the underflow equation. From Equation 2.6, we have that $\oplus x(\mathsf{cbef}_{k,i}(j)) \geq \oplus y(j) - \theta(e)$. Since $\oplus x$ is a monotone function, we have that

$$\mathsf{cbef}_{k,i}(j) \geq x^{-1}(\oplus y(j) - \theta(e)) \tag{2.7}$$

According to Property 2.1, there must be a finite number of firings of the producer between every two firings of the consumer. Hence, function $\mathsf{cbef}_{k,i}$ should satisfy

$$\forall j \in \mathbb{N}_{>0} : \mathsf{cbef}_{k,i}(j) - \mathsf{cbef}_{k,i}(j-1) \text{ is bounded} \tag{2.8}$$

Since $\mathsf{cbef}_{k,i}(\mathsf{cbef}_{i,k}(j)) < j$, there is no activation relation that satisfies both the overflow and the underflow analyses if $\exists j$:

$$x^{-1}(\oplus y(y^{-1}(\oplus x(j) + \theta(e) - \delta(e))) - \theta(e)) \geq j$$

A similar condition can be deduced using $\mathsf{cbef}_{i,k}(\mathsf{cbef}_{k,i}(j)) < j$.

Example 2.12. Let $e = (p_1, p_2, x, 2^\omega)$ be a channel such that x is equal to the *Fibonacci word*. The Fibonacci word is an *aperiodic* infinite binary sequence and cannot hence modeled in the (C)SDF model. If $s_0 = 0$, $s_1 = 0\ 1$, and $s_n = s_{n-1}.s_{n-2}$, then the Fibonacci word is the limit s_∞.

If $\Phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio, then we have that $x(j) = 2 + \lfloor j\Phi \rfloor - \lfloor (j+1)\Phi \rfloor$, $y^{-1}(j) = \left\lceil \frac{j}{2} \right\rceil$, $x^{-1}(j) = \lfloor j\Phi^2 \rfloor$, and $\oplus x(j) = 2j + 1 - \lfloor (j+1)\Phi \rfloor$. According to Equations 2.4 and 2.7, we must have

$$\mathsf{cbef}_{2,1}(j) \geq \lfloor (2j - \theta(e))\Phi^2 \rfloor \tag{2.9}$$

$$\mathsf{cbef}_{1,2}(j) \geq \left\lceil \frac{2j + 1 - \lfloor (j+1)\Phi \rfloor + \theta(e) - \delta(e)}{2} \right\rceil \tag{2.10}$$

A possible schedule of the two actors that could satisfy the previous equations is presented in Listing 2.1. For this schedule, $\mathsf{cbef}_{1,2}(j) = \left\lfloor \frac{j}{2\Phi^2} \right\rfloor$ and $\mathsf{cbef}_{2,1}(j) = \lfloor 2\Phi^2 j \rfloor$. Equation 2.9 is satisfied if $\theta(e) \geq 0$. For $\theta(e) = 0$, Equation 2.10 is satisfied if $\delta(e) \geq 2$. Hence, $\delta(e) = 2$ is the minimum buffer size when $\theta(e) = 0$.

Listing 2.1: A bounded and complete schedule of Example 2.12 .

```
j=1;  k=0;
while(true) do{
     for(i=k+1;i ≤ ⌊2Φ²j⌋ ;i++){ k++; code of p1 }
     code of p2
     j++;
}
```

2.3 Affine schedules

In real-time scheduling of dataflow graphs, a dataflow graph is represented as a periodic task set where each actor is mapped to a periodic task with scheduling parameters (period, phase, deadline, priority, etc). Tasks cannot self-suspend and auto-concurrency is disabled. A periodic task set can be abstracted by a specific class of activation-related schedules, called affine (or strictly periodic) schedules.

2.3.1 Affine relations

Affine transformations of abstract clocks were introduced in [168] (Definition 1.4, p. 22). In this section, we will approach them differently using ultimately periodic binary sequences. Activation clocks $\hat{p}_i$ and $\hat{p}_k$ are (n, φ, d)-affine-related, with $n, d \in \mathbb{N}_{>0}$ and $\varphi \in \mathbb{Z}$, if in case φ is positive (resp. negative), clock $\hat{p}_i$ is obtained by counting each nth instant on a referential abstract clock $\hat{c}$ starting from the first (resp. $(-\varphi+1)^{th}$) instant; while clock $\hat{p}_k$ is obtained by counting each dth instant on $\hat{c}$ starting from the $(\varphi+1)^{th}$ (resp. first) instant. Hence, we have $\forall j \in \mathbb{N}_{>0}$ (j denotes the jth tag in $\hat{c}$)

$$\hat{p}_i(j) = \begin{cases} 1 & \textit{if } \exists t \in \mathbb{N} : j = nt + \max\{1, -\varphi+1\} \\ 0 & \textit{otherwise} \end{cases}$$

and

$$\hat{p}_k(j) = \begin{cases} 1 & \textit{if } \exists t \in \mathbb{N} : j = nt + \max\{1, \varphi+1\} \\ 0 & \textit{otherwise} \end{cases}$$

Figure 2.9 represents a $(3, -4, 5)$-affine relation.

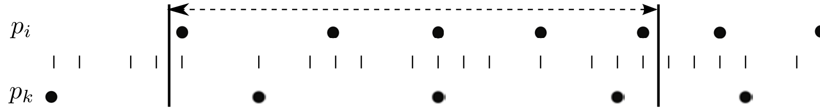


Figure 2.9: A $(3, -4, 5)-$affine relation.

Property 2.5. $\hat{p}_i$ and $\hat{p}_k$ are ultimately periodic binary sequences.

Proof: We have that $\forall j \in \mathbb{N}_{>0} : \hat{p}_i(j + \frac{nd}{\gcd\{n,d\}}) = \hat{p}_i(j)$ and $\hat{p}_k(j + \frac{nd}{\gcd\{n,d\}}) = \hat{p}_k(j)$. □

Therefore, there is a positioning pattern that repeats infinitely, as depicted in Figure 2.9. It consists of $\frac{d}{\gcd\{n,d\}}$ activations of p_i and $\frac{n}{\gcd\{n,d\}}$ activations of p_k. Thus, parameters n and d of an affine relation encode the ratio between activation rates of actors. For instance, a $(1, \varphi, 2)$-affine relation means that p_i is twice as fast as actor p_k. Parameter φ encodes the difference between the first start times of the two actors.

Property 2.6. If p_i and p_k are (n, φ, d)-affine-related, then p_k and p_i are $(d, -\varphi, n)$-affine-related.

In the sequel, φ is supposed to be positive (unless it is stated otherwise). When φ is negative, it is sufficient to reverse the affine relation and then obtain the desired property.

Property 2.7. An affine relation contains synchronous activations if and only if $\gcd\{n, d\}$ divides φ.

Proof: If $p_i[j] = p_k[j']$, then $\hat{p}_i(nj-n+1) = \hat{p}_k(dj'-d+\varphi+1) = 1$ and $nj - dj' = \varphi + n - d$. We have two cases:
(i) $\gcd\{n, d\} | \varphi$: the diophantine equation $\frac{n}{\gcd\{n,d\}} j - \frac{d}{\gcd\{n,d\}} j' = \frac{\varphi+n-d}{\gcd\{n,d\}}$ have an infinite number of solutions. If the first synchronization occurs at (j_1, j'_1), then the lth

synchronization occurs at $(j_1 + \frac{(l-1)d}{\gcd\{n,d\}}, j'_1 + \frac{(l-1)n}{\gcd\{n,d\}})$. Hence, there is exactly one synchronization per positioning pattern instance.
(ii) $\gcd\{n,d\} \not| \varphi$: Equation $nj - dj' = \varphi + n - d$ has no solution. □

Since we are not interested in tags where both clocks are absent, we define an affine relation as an ultimately periodic activation relation $[s_{i,k} = u_i v_i^\omega, s_{k,i} = u_k v_k^\omega]$. We have that (assuming that φ is positive)

- $|u_i| = |u_k| = \lceil \frac{\varphi}{n} \rceil$; $\forall j \leq |u_i| : u_i(j) = 1$ and $u_k(j) = 0$.
- According to Property 2.7,

$$|v_i| = |v_k| = \frac{n+d}{\gcd\{n,d\}} - \begin{cases} 1 & \textit{if } \gcd\{n,d\}|\varphi \\ 0 & \textit{otherwise} \end{cases}$$

- $\|v_i\| = \frac{d}{\gcd\{n,d\}}$ and $\|v_k\| = \frac{n}{\gcd\{n,d\}}$.

Example 2.13. For the affine relation in Figure 2.9, we have that $s_{i,k} = 0(1\ 0\ 1\ 1\ 1\ 0\ 1)^\omega$ and $s_{k,i} = 1(0\ 1\ 0\ 1\ 0\ 1\ 0)^\omega$.

We have also that $\mathsf{sprd}_{i,k}(j) = \max\{0, \left\lceil \frac{n(j-1)-\varphi}{d} \right\rceil\}$, $\mathsf{prd}_{i,k}(j) = \max\{0, \left\lfloor \frac{n(j-1)-\varphi}{d} \right\rfloor + 1\}$, $\mathsf{ssuc}_{i,k}(j) = \mathsf{prd}_{i,k}(j) + 1$, and $\mathsf{suc}_{i,k}(j) = \mathsf{sprd}_{i,k}(j) + 1$. To obtain $\mathsf{sprd}_{k,i}$, $\mathsf{prd}_{k,i}$, $\mathsf{ssuc}_{k,i}$, and $\mathsf{suc}_{k,i}$, we need just to reverse the affine relation. Let $\xi(j)$ be the number of synchronizations until and including $p_i[j]$, and $p_i[j_1]$ be the first job of p_i synchronized with a job of p_k. So, we have that $s_{i,k}^{-1}(j) = j + \mathsf{prd}_{i,k}(j) - \xi(j)$ and

$$\xi(j) = \begin{cases} 0 & \textit{if } \gcd\{n,d\} \not| \varphi \\ \max\{0, \left\lfloor \frac{(j-j_1)\gcd\{n,d\}}{d} \right\rfloor + 1\} & \textit{otherwise} \end{cases}$$

Canonical form

An infinite number of affine transformations can result in the same activation relation. For instance, (n, φ, d) and $(cn, c\varphi, cd)$ (where c is an integer constant) result in the same activation relation. A canonical form is therefore proposed in [167]. Let $g = \gcd\{n,d\}$. So,

(i) $g|\varphi$: $(n, \varphi, d) \equiv (\frac{n}{g}, \frac{\varphi}{g}, \frac{d}{g})$.

(ii) $g \not| \varphi$: $(n, \varphi, d) \equiv (2\frac{n}{g}, 2\left[\frac{\varphi}{g}\right] + 1, 2\frac{d}{g})$.

The first case is obvious; the second case needs a proof. We present here a much simpler proof than the one presented in [167].

Proof: Let $(n, \varphi, d) \equiv (n', \varphi', d')$ with $n' = \frac{mn}{g}$ and $d' = \frac{md}{g}$. We need to find the minimum integers m $(m \geq 1)$ and φ' such that both affine transformations result in the same activation relation; hence in the same $\mathsf{sprd}_{i,k}$ and $\mathsf{ssuc}_{i,k}$.

$\mathsf{sprd}_{i,k}(j) = \left\lceil \frac{n'(j-1)-\varphi'}{d'} \right\rceil = \left\lceil \frac{n(j-1)-\varphi}{d} + \frac{\varphi}{d} - \frac{g\varphi'}{md} \right\rceil$. The two affine transformations are equivalent only if $\left\lceil \frac{n(j-1)-\varphi}{d} \right\rceil = \left\lceil \frac{n(j-1)-\varphi}{d} + \frac{\varphi}{d} - \frac{g\varphi'}{md} \right\rceil$. Let $d_0 \geq 1$ and $d_1 \geq 1$ be the minimum integers such that $d|n(j-1)-\varphi-d_0$ and $d|n(j-1)-\varphi+d_1$, respectively. Note that d_0 and d_1 are greater or equal to 1 because the affine relation does not contain any synchronous activations (since $g \not| \varphi$). Hence, we must have $-d_0 < \varphi - \frac{g}{m}\varphi' \leq d_1$. Following the same process with $\mathsf{ssuc}_{i,k}$, we should have the constraint $-d_0 \leq \varphi - \frac{g}{m}\varphi' < d_1$. Therefore, $\frac{\varphi-d_1}{g} < \frac{\varphi'}{m} < \frac{\varphi+d_0}{g}$. So, we must have $\frac{\varphi-\min_j d_1}{g} < \frac{\varphi'}{m} < \frac{\varphi+\min_j d_0}{g}$. Since $\min_j d_0 + \min_j d_1 = g$, bounds $\frac{\varphi-\min_j d_1}{g}$ and $\frac{\varphi+\min_j d_0}{g}$ are two successive integers. Hence, the minimum value of m is 2 and the minimum value of φ' is $2\left[\frac{\varphi}{g}\right]+1$. □

2.3.2 Consistency

An affine schedule consists of a set of affine relations. Let G_r be the graph of affine relations.

Proposition 2.7. A closed walk $p_1 \xrightarrow{(n_1,\varphi_1,d_1)} p_2 \to \cdots \to p_m \xrightarrow{(n_m,\varphi_m,d_m)} p_1$ is consistent only if

$$\prod_{i=1}^{m} n_i = \prod_{i=1}^{m} d_i \tag{2.11}$$

Proof: Since an affine schedule is also an ultimately periodic schedule, a closed walk in G_r is consistent only if Proposition 2.8 is satisfied; i.e. $\prod_{i=1}^{m} \|v_{i,k}\| = \prod_{i=1}^{m} \|v_{k,i}\|$ with $k = i \mod m+1$. But, $\|v_{i,k}\| = \frac{d_i}{\gcd\{n_i,d_i\}}$ and $\|v_{k,i}\| = \frac{n_i}{\gcd\{n_i,d_i\}}$. □

Proposition 2.8. A closed walk $p_1 \xrightarrow{(n_1,\varphi_1,d_1)} p_2 \to \cdots \to p_m \xrightarrow{(n_m,\varphi_m,d_m)} p_1$ is consistent if Equation 2.11 is satisfied and

$$\sum_{i=1}^{m}(\prod_{j=1}^{i-1} d_j)(\prod_{j=i+1}^{m} n_j)\varphi_i = 0 \tag{2.12}$$

Proof: Let us put $X_m = \prod_{i=1}^{m} n_i$, $Y_m = \prod_{i=1}^{m} d_i$, and $Z_m = \sum_{i=1}^{m}(\prod_{j=1}^{i-1} d_j)(\prod_{j=i+1}^{m} n_j)\varphi_i$. We have two cases.
(i) $\psi[j]$ is true: In this case, $\mathsf{prd}_{1,2}(j) = \frac{X_1 j - Z_1}{Y_1} - \frac{X_1}{Y_1} + 1$, $\mathsf{prd}_{1,2,3}(j) = \frac{X_2 j - Z_2}{Y_2} - \frac{X_2}{Y_2} + 1$, etc. Since $X_m = Y_m$, we have that $\mathsf{prd}_\psi(j) = j - \frac{Z_m}{Y_m}$. According to Proposition 2.1, $\mathsf{prd}_\psi(j) = j$; hence $Z_m = 0$. Similar process can be done for suc_ψ.
(ii) $\psi[j]$ is false: Since $\mathsf{prd}_\psi(j) < j - \frac{Z_m}{Y_m} < \mathsf{suc}_\psi(j)$, condition $Z_m = 0$ is a sufficient condition for consistency. □

Example 2.14. Proposition 2.8 is a sufficient but non-necessary condition for consistency of a cycle. If $p_1 \stackrel{(3,5,5)}{\longrightarrow} p_2 \stackrel{(4,-8,3)}{\longrightarrow} p_3 \stackrel{(n_3,\varphi_3,d_3)}{\longrightarrow} p_1$, then according to Equations 2.11 and 2.12 we have that $4n_3 = 5d_3$ and $4n_3 = 3\varphi_3$. So, $(n_3, \varphi_3, d_3) \equiv (15k, 20k, 12k)$ for $k \geq 1$. Since $\gcd\{15k, 12k\} \not| 20k$, the canonical form is $(10, 13, 8)$. However, Equation 2.12 does not hold when $(n_3, \varphi_3, d_3) = (10, 13, 8)$. The affine schedule is also consistent when $(n_3, \varphi_3, d_3) = (10, 11, 8)$, as illustrated in Figure 2.10. Equation 2.12 does not also hold in this case. All the affine transformations that are equivalent with $(10, 11, 8)$ can be written as $(5k, \varphi, 4k)$ with $5k < \varphi < 6k$. All these affine transformations cannot satisfy Equation 2.12.

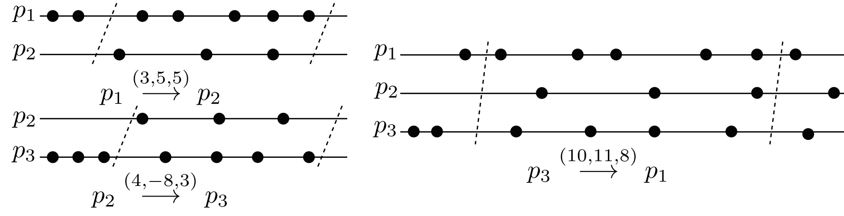


Figure 2.10: A consistent affine schedule.

Property 2.8. Equation 2.12 is a necessary condition if the following equation system admits a solution.

$$\Gamma \vec{x} = \vec{\varphi} \tag{2.13}$$

where $\vec{\varphi}(i) = \varphi_i$, and for every affine relation $p_i \stackrel{(n_i,\varphi_i,d_i)}{\longrightarrow} p_k$ with $k = (i \mod m) + 1$, we have that $\Gamma_{i,i} = n_i$ and $\Gamma_{i,k} = -d_i$.

Proof: Firstly, if Equation 2.13 admits a solution, then it admits an infinite number of solutions since $\det(\Gamma) = X_m - Y_m = 0$. If $\vec{x}$ is a solution, then $\forall i = 1, m : n_i\vec{x}(i) - d_i\vec{x}(k) = \varphi_i$ such that $k = (i \mod m) + 1$. Hence, $\mathsf{prd}_{i,k}(\vec{x}(i) + 1) = \mathsf{suc}_{i,k}(\vec{x}(i) + 1) = \vec{x}(k) + 1$. So, $p_i[\vec{x}(i) + 1] = p_k[\vec{x}(k) + 1]$. Thus, $\psi[\vec{x}(1) + 1]$ is true. According to the proof of Proposition 2.8, Z_m must be equal to zero. □

In real-time scheduling of dataflow graphs, we need to have a global referential clock. Therefore, Proposition 2.8 must hold for every simple cycle in the graph of affine relations.

Proposition 2.9. Let ψ, ψ', ψ'' be three simple cycles such that $\psi = \psi' \uplus \psi''$ and $\psi' \cup \psi''$ forms a continuous path. If Proposition 2.8 holds for ψ' and ψ'', then it holds for ψ.

Proof: According to Property 2.4 and since an affine schedule is also an ultimately periodic schedule, we have that Equation 2.11 holds for cycle ψ. Since the set of edges in ψ' that have opposites in ψ'' forms a contiguous path, we can put $\psi' = \psi_1 + \psi_*$, $\psi'' = \psi_2 + \psi_*^{-1}$, and $\psi = \psi_1 + \psi_2$. To ease the notations, we illustrate the proof for Equation

2.12 on an example. Let $\psi_* = p_2 \overset{(n,\varphi,d)}{\longrightarrow} p_3 \overset{(n',\varphi',d')}{\longrightarrow} p_4$, $\psi_1 = p_4 \overset{(n_4,\varphi_4,d_4)}{\longrightarrow} p_1 \overset{(n_1,\varphi_1,d_1)}{\longrightarrow} p_2$, and $\psi_2 = p_2 \overset{(n_2,\varphi_2,d_2)}{\longrightarrow} p_5 \overset{(n_5,\varphi_5,d_5)}{\longrightarrow} p_4$.

So, we have that $Z_{\psi'} = n_1\varphi_4 nn' + d_4\varphi_1 nn' + d_4 d_1 \varphi n' + d_4 d_1 \varphi' d = 0$ and $Z_{\psi''} = n_5\varphi_2 d'd + d_2\varphi_5 d'd - d_2 d_5 \varphi' d - d_2 d_5 \varphi n' = 0$. We then eliminate common terms: $d_2 d_5 Z_{\psi'} + d_4 d_1 Z_{\psi''} = n_5 d_4 d_1 \varphi_2 dd' + d_2 d_4 d_1 \varphi_5 dd' + n_1 d_2 d_5 \varphi_4 nn' + d_4 d_2 d_5 \varphi_1 nn' = 0$. Using $\frac{nn'}{dd'} = \frac{d_4 d_1}{n_4 n_1}$ (Equation 2.11 on cycle ψ'), we obtain $\frac{d_2 d_5 Z_{\psi'} + d_4 d_1 Z_{\psi''}}{dd'} = Z_\psi = 0$.

Generally, if X_ψ is the product of parameters n of affine relations on walk ψ and Y_ψ is the product of parameters d of affine relations on walk ψ, then

$$Y_{\psi_1} Z_{\psi''} + Y_{\psi_2} Z_{\psi'} = Y_{\psi_*} Z_\psi$$

□

Proposition 2.10 (Consistency of affine schedules)**.** An affine schedule is consistent if Equations 2.11 and 2.12 hold for every fundamental cycle in the graph of affine relations.

A fundamental cycle is an undirected cycle that is obtained when adding an edge to the spanning tree (or the spanning forest if G_r is not connected) of the graph of affine relations. There is a one-to-one correspondence between fundamental cycles and edges not in the spanning tree. if $G_r = (P, R)$ is connected, then there are $|R| - N + 1$ fundamental cycles. Fundamental cycles form a basis for the cycle space; i.e. any undirected cycle in the graph can be expressed as the symmetric difference of some fundamental cycles. According to Proposition 2.9, we need to check consistency of only cycles in a cycle basis. Finding a cycle basis is a well-known problem in graph theory [110] that can be solved in polynomial time. Finding the minimum cycle basis (i.e. the total number of edges in the cycle basis is minimum) could be interesting to reduce the length of constraints (i.e. Equation 2.12).

2.3.3 Fixed-priority schedules

In FP scheduling, each actor p_i has a constant priority sequence $\omega_i = (w_i)^\omega$. We assume that each actor can have a distinguished priority. At each instant of the execution, a FP scheduler chooses the task with the highest priority among all activated tasks to execute on a given processor. In case of partitioned scheduling, each actor is permanently allocated to processor number ν_i; while in global scheduling, a job can migrate from one processor to another. We will show in the next chapter how to compute the scheduling parameters w_i and ν_i of each actor. In this section, we present how to compute a FP abstract schedule using Linear Integer Programming (ILP). The abstract schedule consists of an affine relation between every two adjacent (i.e. communicating) actors. The abstract schedule should exclude overflow and underflow exceptions.

A Overflow analysis

Let $e = (p_i, p_k, x, y)$ be a channel between the (n, φ, d)-affine-related actors p_i and p_k. The affine relation is under the canonical form. Firstly, we need to compute function

$\mathsf{cbef}_{i,k}$ that corresponds to a FP scheduling policy without considering the physical timing. Figures 2.11 illustrates for all possible cases which last job of p_k certainly reads all its needed tokens from channel e before $p_i[j]$ writes any token on that channel. Let $j' = \mathsf{prd}_{i,k}(j)$ and $j'' = \mathsf{cbef}_{i,k}(j)$.

(i) *Partitioned scheduling*:

a. $w_i < w_k$, $\nu_i = \nu_k$, and $p_i[j] \neq p_k[j']$: When job $p_i[j]$ is released, it may be possible that $p_k[j']$ has not yet finished reading from channel e. In this case, $p_i[j]$ preempts $p_k[j']$ because p_i has the highest priority and both jobs are allocated to the same processor. Thus, we are only certain that $p_k[j'-1]$ has already finished. A safe approximation is hence to put $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$.

b. $w_i < w_k$, $\nu_i = \nu_k$, and $p_i[j] = p_k[j']$: jobs $p_i[j]$ and $p_k[j']$ are released simultaneously to execute on the same processor. Thus, $p_i[j]$ will execute first because it has the highest priority. Hence, $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$.

c. $w_i < w_k$, $\nu_i \neq \nu_k$, and $p_i[j] \neq p_k[j']$: As in case (a), $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$.

d. $w_i < w_k$, $\nu_i \neq \nu_k$, and $p_i[j] = p_k[j']$: As in case (b), $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$.

e. $w_i > w_k$, $\nu_i = \nu_k$, and $p_i[j] \neq p_k[j']$: Since $p_k[j']$ has a higher priority than $p_i[j]$ and $p_k[j']$ is released first, we are certain that $p_k[j']$ ends before $p_i[j]$ starts writing any token. Hence, $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j)$.

f. $w_i > w_k$, $\nu_i = \nu_k$, and $p_i[j] = p_k[j']$: As in the previous case, $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j)$.

g. $w_i > w_k$, $\nu_i \neq \nu_k$, and $p_i[j] \neq p_k[j']$: Though $p_k[j']$ is released before $p_i[j]$ and $p_k[j']$ has the highest priority, we are not certain that $p_k[j']$ completes reading from channel e before $p_i[j]$ starts writing. Indeed, $p_i[j]$ and $p_k[j']$ can execute in parallel with each other because they are allocated to different processors. A safe approximation is hence $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$.

h. $w_i > w_k$, $\nu_i \neq \nu_k$, and $p_i[j] = p_k[j']$: $p_i[j]$ can execute in parallel with $p_k[j']$. Thus, $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$.

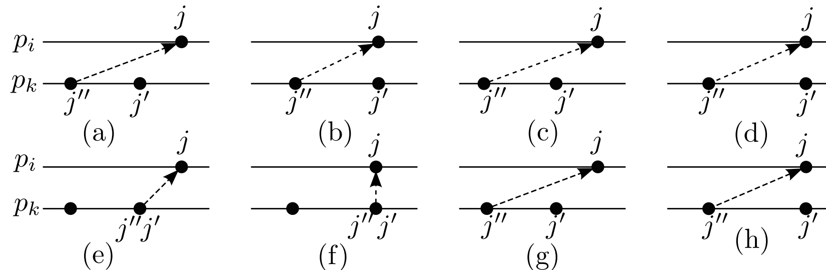


Figure 2.11: Function $\mathsf{cbef}_{i,k}$: partitioned fixed-priority scheduling.

To sum up, it is certain that the last job of p_k released before or simultaneously with $p_i[j]$, completes reading from channel e before $p_i[j]$ starts writing on e only if p_k has a higher priority than p_i and both actors are allocated to the same processor. Therefore,

$$\mathsf{cbef}_{i,k}(j) = \begin{cases} \max\{0, \mathsf{prd}_{i,k}(j) - 1\} & \textit{if } w_i < w_k \vee \nu_i \neq \nu_k \\ \mathsf{prd}_{i,k}(j) & \textit{otherwise} \end{cases}$$

(ii) *Global scheduling*: It is possible that there are enough resources to execute $p_i[j]$ and $p_k[j']$ in parallel with each other. Hence, a safe approximation is to take $\mathsf{cbef}_{i,k}(j) = \max\{0, \mathsf{prd}_{i,k}(j) - 1\}$.

Channel e is overflow-free if Equation 2.3 is satisfied. Unfortunately, that equation is not a linear constraint. Similarly to real-time calculus, we approximate the number of accumulated tokens by upper and lower bound curves. However, the curves are time-independent. If $\oplus x^u$ is the upper bound curve of $\oplus x$, $\oplus y^l$ is the lower bound curve of $\oplus y$, and $\mathsf{cbef}^l_{i,k}$ is the lower bound curve of $\mathsf{cbef}_{i,k}$, then the over-approximated number of accumulated tokens is given as

$$X^u(j) = \theta(e) + \oplus x^u(j) - \oplus y^l(\mathsf{cbef}^l_{i,k}(j))$$

In order to compute $\mathsf{cbef}^l_{i,k}$, we need to compute the lower bound of $\left\lfloor \frac{n(j-1)-\varphi}{d} \right\rfloor$ which could be taken as the linear bound $\frac{n}{d}j - \frac{\varphi+n+d-1}{d}$. Since an actor produces or consumes a finite number of tokens at each firing, the cumulative function of a rate function x can be linearly bounded. Suppose that we have $\forall j \in \mathbb{N} : \oplus x^l(j) = a_x j + \lambda^l_x \leq \oplus x(j) \leq \oplus x^u(j) = a_x j + \lambda^u_x$ with $a_x \in \mathbb{Q}_{>0}$. If $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$, then we obtain

$$X^u(j) = (a_x - a_y \frac{n}{d})j + \theta(e) + \lambda^u_x - \lambda^l_y + a_y \frac{\varphi + n + d - 1}{d}$$

The number of accumulated tokens is bounded only if $a_x - a_y \frac{n}{d} = 0$. Thus, the boundedness criterion is

$$\frac{n}{d} = \frac{a_x}{a_y} \tag{2.14}$$

Since n and d can be deduced from the boundedness criterion, the overflow equation can be written as a simple linear constraint

$$\theta(e) - \delta(e) + \frac{a_y}{d}\varphi \leq \lambda^l_y - \lambda^u_x - a_y \frac{n + d - 1}{d} \tag{2.15}$$

where φ is the free integer variable. The size of the channel and the number of its initial tokens can be either predefined or free variables. If $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j)$, we obtain the following linear constraint.

$$\theta(e) - \delta(e) + \frac{a_y}{d}\varphi \leq \lambda^l_y - \lambda^u_x - a_y \frac{n - 1}{d} \tag{2.16}$$

B Underflow analysis

The underflow analysis is dual to the overflow analysis. We first need to compute function $\mathsf{cbef}_{k,i}$ that corresponds to a FP scheduling policy without considering the physical timing. Figure 2.12 illustrates for some possible cases which last job of p_i certainly writes all its results on channel e before $p_k[j]$ reads any token from that channel.

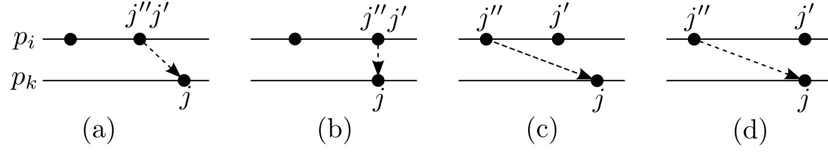


Figure 2.12: Function $\mathsf{cbef}_{k,i}$: partitioned fixed-priority scheduling.

In partitioned scheduling, $\mathsf{cbef}_{k,i}(j) = \mathsf{prd}_{k,i}(j)$ if $w_i < w_k$ and $\nu_i = \nu_k$ (cases (a) and (b)); and $\mathsf{cbef}_{k,i}(j) = \mathsf{prd}_{k,i}(j) - 1$ otherwise. In global scheduling, $\mathsf{cbef}_{k,i}(j) = \mathsf{prd}_{k,i}(j) - 1$ (cases (c) and (d)). The number of accumulated token in channel e can be under-approximated as

$$Y^l(j) = \theta(e) + \oplus x^l(\mathsf{cbef}^l_{k,i}(j)) - \oplus y^u(j)$$

If $\mathsf{cbef}_{k,i}(j) = \mathsf{prd}_{k,i}(j) - 1$, then $\mathsf{cbef}_{k,i}(j)$ can be linearly lower bounded by $\frac{d}{n}j + \frac{\varphi-n-d+1}{n}$. Hence,

$$Y^l(j) = (a_x\frac{d}{n} - a_y)j + \theta(e) + \lambda^l_x - \lambda^u_y + a_x\frac{\varphi - n - d + 1}{n}$$

But, $a_x\frac{d}{n} - a_y = 0$ (the boundedness criterion). Therefore, the underflow equation can be written as the following linear constraint.

$$\theta(e) + \frac{a_y}{d}\varphi \geq \lambda^u_y - \lambda^l_x + a_y\frac{n + d - 1}{d} \tag{2.17}$$

When $\mathsf{cbef}_{k,i}(j) = \mathsf{prd}_{k,i}(j)$, the linear underflow constraint is given as.

$$\theta(e) + \frac{a_y}{d}\varphi \geq \lambda^u_y - \lambda^l_x + a_y\frac{d - 1}{d} \tag{2.18}$$

If we combine the overflow equation with the underflow equation, then we can compute an approximate size of the channel as illustrated in Table 2.1 where $\lambda = (\lambda^u_y + \lambda^u_x) - (\lambda^l_y + \lambda^l_x)$. These approximate sizes are useful when computing the scheduling parameters. For instance, the gain in storage space that comes from switching the priorities of two actors allocated to the same processor is approximatively equal to $a_y|\frac{d-n}{n}|$.

	$\nu_i = \nu_k$	$\nu_i \neq \nu_k$
$w_i < w_k$	$\lambda + a_y \frac{n+2d-2}{d}$	$\lambda + a_y \frac{2n+2d-2}{d}$
$w_i > w_k$	$\lambda + a_y \frac{2n+d-2}{d}$	$\lambda + a_y \frac{2n+2d-2}{d}$

Table 2.1: Approximate buffer sizes in fixed-priority scheduling.

C Algorithm

If the graph of affine relations is acyclic, then every affine relation can be computed independently of the other relations. Parameters n and d can be deduced from the boundedness criterion; while parameter φ is computed (using an enumerative solution or the approximate overflow and underflow equations) so that the total sum of sizes of the channels between p_i and p_k is minimized.

When the graph of affine relations is cyclic, consistency of the schedule must be considered. Firstly, we use Equation 2.14 to deduce parameters n and d of every affine relation, then Equation 2.11 is applied on every fundamental cycle in G_r. To compute the parameters φ, we construct an integer linear program by applying overflow and underflow linear constraints on channels; and Equation 2.12 on fundamental cycles. The objective function of the linear program is to minimize the buffering requirements. From the overflow and underflow equations, we can notice that $\lim\limits_{|\varphi| \to +\infty} \frac{\delta(e)}{|\varphi|} = \frac{a_y}{d}$. This indicates that the best values of φ are in the neighborhood of 0.

The number of initial tokens in channels could be insufficient and an underflow-free affine schedule does not hence exist. It is also possible to not find an underflow-free schedule because of the conservative approximation made in the overflow and underflow analyses. One option is to not specify the numbers of initial tokens and so let the scheduling tool compute the appropriate values. Sizes obtained by the solution of the linear program are a safe approximation of the actual sizes. Therefore, they may be recomputed after obtaining the affine relations. The minimum number of initial tokens is given by $\theta(e) = |\min\{0, \min\limits_j \{\oplus x(\mathsf{cbef}_{k,i}(j)) - \oplus y(j)\}\}|$ and the minimum size of the channel is given by $\delta(e) = \max\limits_j \{\theta(e) + \oplus x(j) - \oplus y(\mathsf{cbef}_{i,k}(j))\}$.

As shown in the previous chapter, retiming techniques can be used to redistribute the initial tokens in order to improve the throughput. If the production and consumption rates are constant, then retiming can be easily expressed as linear constraints. If $e = (p_i, p_k, a^\omega, b^\omega)$ and $\vec{z}$ is the retiming vector, then we have to add the linear constraint $\theta(e) = \theta'(e) + a\vec{z}(i) - b\vec{z}(k)$ where $\theta'(e)$ is the user-provided number of initial tokens in the channel.

2.3.4 EDF schedules

In EDF scheduling, the job with the earliest absolute deadline has the highest priority. If two jobs $p_i[j]$ and $p_k[j']$ have the same absolute deadlines, then the job which is released first gets the highest priority. However, if the two jobs are synchronous, then we use the actors' ID to break the tie. In this section, we show how to compute the

affine schedule when the dataflow graph is modeled as an implicit periodic task set. We also show the impact of the totally ordered communication strategy on the buffering requirements.

A Implicit task model

Each job $p[j]$ has an absolute deadline that has the same tag as the next release $p[j+1]$. Let $e = (p_i, p_k, x, y)$ be a channel between two (n, φ, d)-affine-related actors. Regarding the overflow analysis, we need to compute function $\mathsf{cbef}_{i,k}$ that corresponds to the EDF scheduling of implicit task sets. Figures 2.13 illustrates for some possible cases which last job of p_k certainly reads all its needed tokens from channel e before $p_i[j]$ writes any token on that channel. Let $j' = \mathsf{prd}_{i,k}(j)$ and $j'' = \mathsf{cbef}_{i,k}(j)$.

(i) *Partitioned scheduling* ($\nu_i = \nu_k$):

a,b. $p_i[j] \neq p_k[j']$ and $p_k[j'+1] \leq p_i[j+1]$: According to the EDF policy, $p_k[j']$ has a higher priority than $p_i[j]$. Since $p_k[j']$ is released before $p_i[j]$ and the two actors are allocated to the same processor, $p_k[j']$ will end before $p_i[j]$ starts executing. Hence, $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j)$.

c. $p_i[j] \neq p_k[j']$ and $p_i[j+1] < p_k[j'+1]$: job $p_k[j']$ is released first; but it can be preempted by $p_i[j]$ since this latter has the highest priority. Hence, it is safe to take $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$.

d. $p_i[j] = p_k[j']$ and $p_k[j'+1] < p_i[j+1]$: $p_k[j']$ has a higher priority than $p_i[j]$. Therefore, $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j)$.

e. $p_i[j] = p_k[j']$ and $p_i[j+1] = p_k[j'+1]$: To break the tie, we need to look at the actors' ID. If $\mathsf{ID}(p_k) < \mathsf{ID}(p_i)$, then $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j)$; else $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$.

f. $p_i[j] = p_k[j']$ and $p_i[j+1] < p_k[j'+1]$: job $p_i[j]$ has a higher priority than $p_k[j']$. Hence, $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$.

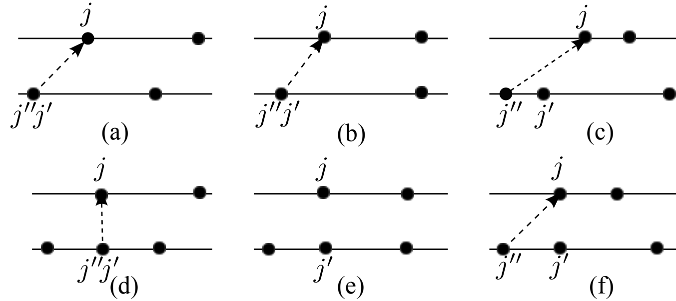


Figure 2.13: Function $\mathsf{cbef}_{i,k}$: partitioned EDF scheduling, $\nu_i = \nu_k$.

There are only three cases in which $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$ (cases (c), (e), and (f)) and which are possible only if $\frac{d-n}{\gcd\{n,d\}} \geq 2$ or if $n = d \wedge \mathit{ID}(p_i) < \mathit{ID}(p_k)$.

(ii) *Partitioned scheduling ($\nu_i \neq \nu_k$) or global scheduling*: In these cases, priorities do not guarantee any precedences between jobs as shown in the FP scheduling. For instance, in Figure 2.13, case (a), jobs $p_k[j']$ and $p_i[j]$ can execute in parallel with each other. So, for all possible cases, we have that $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$.

The underflow analysis is dual to the overflow analysis. For instance, if $p_k[j] = p_i[j']$, $\nu_i = \nu_k$, and $p_k[j+1] < p_i[j'+1]$, then $\mathsf{cbef}_{k,i}(j) = \mathsf{prd}_{k,i}(j) - 1$. The overflow and underflow linear equations can be computed as in the FP scheduling. Similarly, we can deduce the approximate gain that comes from allocating two actors to the same processor.

B Totally ordered communication

Consider the case when the producer p_i and the consumer p_k are all allocated to the same processor. As illustrated in Figure 2.13 case (c), worst-case overflow and underflow scenarios are considered when there is a potential preemption. Besides reducing the context switching overhead, eliminating preemptions will definitely result in a more accurate analysis and hence less buffering requirements.

The totally ordered communication strategy between p_i and p_k is defined as follows. If $p_i[j]$ is released before $p_k[j']$, then $p_i[j]$ executes entirely before $p_k[j']$. Similarly, if $p_k[j']$ is released before $p_i[j]$, then $p_k[j']$ executes entirely before $p_i[j]$. In the next chapter, we will describe how to implement this communication strategy without using lock-based synchronization mechanisms.

For the overflow analysis, we will have $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j)$ in cases (a,b,c,d) of Figure 2.13 and $\mathsf{cbef}_{i,k}(j) = \mathsf{prd}_{i,k}(j) - 1$ in case (f). We need to look at actors' ID to break the tie in case (e). The difference between the implicit task model and the totally ordered communication strategy lies in case (c); i.e. when $\frac{d-n}{\gcd\{n,d\}} \geq 2$. In this case, the linear overflow and underflow equations are

$$\lambda_y^u - \lambda_x^l + a_y \frac{d-1}{d} \leq \theta(e) + \frac{a_y}{d}\varphi \leq \delta(e) + \lambda_y^l - \lambda_x^u - a_y \frac{n}{d}$$

Compared to the implicit task model, the approximate gain on buffering requirements is hence equal to $a_y \frac{d-1}{d}$. Similarly, if $\frac{n-d}{\gcd\{n,d\}} \geq 2$, then the consumer may preempt the producer. The approximate gain that comes from the totally ordered communication will be equal to $a_x \frac{n-1}{n}$.

One important question about the totally ordered communication strategy is whether imposing scheduling precedences may create deadlocked cycles. The answer is ***no*** because we enforce a scheduling precedence from p_i to p_k only when $\frac{n-d}{\gcd\{n,d\}} \geq 2$. This property cannot be satisfied by all relations on a cycle.

2.4 Specific cases

In this section, we refine our technique for ultimately cyclo-static dataflow graphs. We also extend the approach to consider multichannels, shared storage space, synchronous semantics, etc.

2.4.1 Ultimately cyclo-static dataflow graphs

Definition 2.9 (UCSDF graph)**.** An ultimately cyclo-static dataflow (UCSDF) graph is a static dataflow graph where production and consumption rates are ultimately periodic.

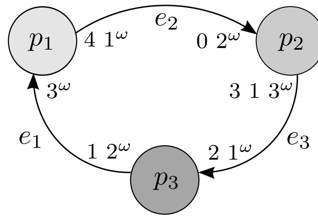


Figure 2.14: An ultimately cyclo-static dataflow graph.

Figure 2.14 represents an UCSDF graph. It is not evident how to transform an UCSDF graph to an equivalent SDF graph *without refactoring the code*. Let us consider a very simple UCSDF graph $G = (\{p_1, p_2\}, e = (p_1, p_2, x = u_1 v_1^\omega, y = u_2 v_2^\omega))$. This graph can be modeled as a (C)SDF graph only if there are two finite integers $j_1 \geq |u_1|$ and $j_2 \geq |u_2|$ such that $\theta(e) + \oplus x(j_1) - \oplus y(j_2) = 0$. For example, if $x = 3\ 0(2\ 1)^\omega$, $y = 3(2\ 0\ 1)^\omega$, and $\theta(e) = 2$, then $j_1 = 2$ and $j_2 = 2$. This way the graph can be partitioned into two disjoint (C)SDF graphs: a deadlocked graph and a live one, as illustrated in Figure 2.15. Note that actor $p_1[1]$ will fire only once (and hence corresponds to the first job of actor p_1) because of the non-consistent self loop.

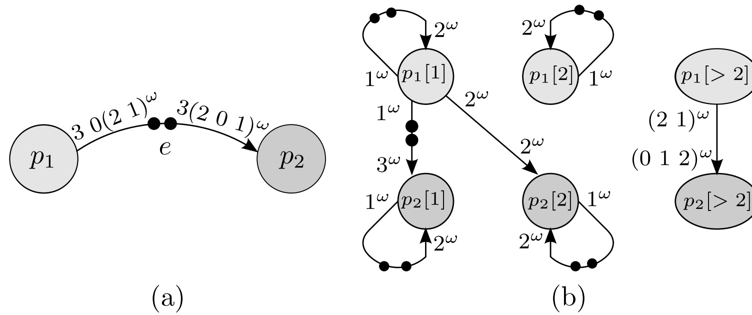


Figure 2.15: (a) An UCSDF graph and (b) its equivalent CSDF graph.

There are some problems in this transformation method: (1) The resulted graph is non-consistent, which hinders the use of analysis tools for (C)SDF graphs. (2) We cannot disable auto-concurrency because this requires to add edges from the deadlocked subgraph into the live one and hence turns this latter into a deadlocked graph. (3) The UCSDF graph in Figure 2.14 cannot be transformed because it is not possible to find three finite integers $j_1 \geq 1$, $j_2 \geq 2$, and $j_3 \geq 1$ that bring the marking to $[0\ 0\ 0]^\mathsf{T}$.

If Γ denotes the reachability graph in Figure 1.2 (p. 27), then the reachability graph

of the UCSDF example (Figure 2.14) is

$$\begin{bmatrix}0\\0\\0\end{bmatrix} \xrightarrow{p_2} \begin{bmatrix}0\\0\\3\end{bmatrix} \xrightarrow{p_3} \begin{bmatrix}1\\0\\1\end{bmatrix} \xrightarrow{p_3} \begin{bmatrix}3\\0\\0\end{bmatrix} \xrightarrow{p_1} \begin{bmatrix}0\\4\\0\end{bmatrix} \xrightarrow{p_2} \Gamma$$

Therefore, there are an infinite number of live and bounded static schedules for the UCSDF example. All these schedules start with the prefix $p_2\ p_3\ p_3\ p_1\ p_2$. Unlike in reachability graphs of (C)SDF graphs, it is possible that the initial marking of an UCSDF graph is not a home marking (i.e. there is no firing sequence that brings the graph to its initial state).

Affine scheduling

Let $x = uv^\omega$ be an ultimately periodic rate function. So, $\oplus x$ is a function that increases by $\|v\|$ every $|v|$ steps. Therefore, the cumulative function can be linearly bounded by $a_x j + \lambda_x^l \leq \oplus x \leq a_x j + \lambda_x^u$ where $a_x = \frac{\|v\|}{|v|}$, $\lambda_x^l = \min\limits_{0 \leq j \leq |u|+|v|} \{\oplus x(j) - \frac{\|v\|}{|v|} j\}$, and $\lambda_x^u = \max\limits_{0 \leq j \leq |u|+|v|} \{\oplus x(j) - \frac{\|v\|}{|v|} j\}$. Figure 2.16 illustrates the linear bounds for $x = 2\ 0\ 1 (2\ 1\ 0\ 2)^\omega$.

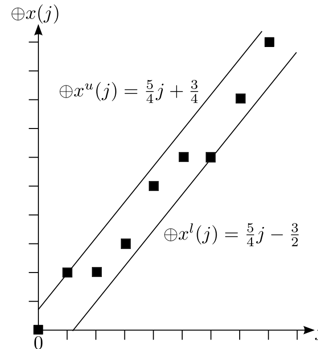


Figure 2.16: Linear bounds of ultimately periodic rates.

If $e = (p_i, p_k, x = u_1 v_1^\omega, y = u_2 v_2^\omega)$ is a channel between p_i and p_k, then the affine relation between the two actors must satisfy the boundedness criterion

$$\frac{n}{d} = \frac{a_x}{a_y} = \frac{\|v_1\|}{|v_1|} \frac{|v_2|}{\|v_2\|}$$

Question: Is there an affine schedule among all the possible static schedules of the UCSDF example?

Firstly, it is worth recalling that the positioning pattern in an affine relation always consists of $\frac{d}{\gcd\{n,d\}}$ activations of p_i and $\frac{n}{\gcd\{n,d\}}$ activations of p_k whatever the parameter φ. Using the boundedness criterion, we may have $p_1 \stackrel{(2,\varphi_1,4)}{\longrightarrow} p_2 \stackrel{(6,\varphi_2,2)}{\longrightarrow} p_3 \stackrel{(4,\varphi_3,6)}{\longrightarrow} p_1$. Let us consider the static schedule $p_2\ p_3\ p_3\ p_1\ p_2 (p_2\ p_3\ p_3\ p_1\ p_3\ p_1)^\omega$. By projecting this

schedule on each pair of the actors, we obtain the activation relations in Figure 2.17. Assuming an EDF scheduling policy with a totally ordered communication strategy on a single processor, we have to find the affine relations that give the same order of firings as in the static schedule. So, the activation relation between p_1 and p_3 corresponds to the affine relation $p_1 \xrightarrow{(6,-7,4)} p_3$. However, there is no affine relation (by enumerating values of φ_1) that can give the same order of firings as in Figure 2.17.(a) because of the prefix. Therefore and for the same reason, there is no underflow-free affine schedule for the chosen scheduling policy. This is why it is better to let the scheduling tool compute the appropriate numbers of initial tokens.

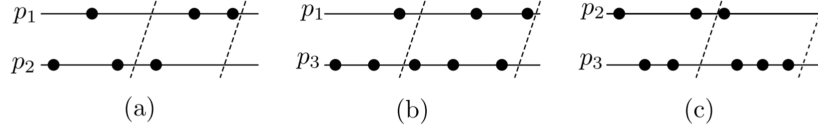


Figure 2.17: Activation relations of the schedule $p_2\ p_3\ p_3\ p_1\ p_2(p_2\ p_3\ p_3\ p_1\ p_3\ p_1)^\omega$.

2.4.2 Multichannels

Let $e = (p_i, p_k, p_l, x, y, z)$ be a channel that relates two producers p_i and p_k (allocated to the same processor) to one consumer p_l. We impose a totally ordered communication strategy between producers; i.e. a job of a producer cannot preempt a job of the other producer. This way, if the affine relation between p_i and p_k is predefined or if all the possible values of the affine relation result in the same order of firings of the producers, then the producers write tokens on channel e in a deterministic way. We have hence a deterministic merge operator.

Like with simple channels, the number of accumulated tokens must be always smaller than or equal to the size of the channel and greater than or equal to zero. The overflow analysis can be written as

$$\forall j \in \mathbb{N}_{>0} : \theta(e) + \oplus x(j) + \oplus y(\mathsf{cbef}_{i,k}(j)) - \oplus z(\mathsf{cbef}_{i,l}(j)) \leq \delta(e)$$

$$\forall j \in \mathbb{N}_{>0} : \theta(e) + \oplus x(\mathsf{cbef}_{k,i}(j)) + \oplus y(j) - \oplus z(\mathsf{cbef}_{k,l}(j)) \leq \delta(e)$$

The number of accumulated tokens can be linearly over-approximated by considering $\oplus x^u$, $\oplus y^u$, $\oplus z^l$, $\mathsf{cbef}^u_{i,k}$, $\mathsf{cbef}^u_{k,i}$, $\mathsf{cbef}^l_{i,l}$, and $\mathsf{cbef}^l_{k,l}$. Dually, the underflow analysis can be written as

$$\forall j \in \mathbb{N}_{>0} : \theta(e) + \oplus x(\mathsf{cbef}_{l,i}(j)) + \oplus y(\mathsf{cbef}_{l,k}(j)) - \oplus z(j) \geq 0$$

The number of accumulated tokens can be linearly under-approximated by considering $\oplus x^l$, $\oplus y^l$, $\oplus z^u$, $\mathsf{cbef}^l_{l,i}$, and $\mathsf{cbef}^l_{l,k}$. If $p_i \xrightarrow{(n_1,\varphi_1,d_1)} p_k \xrightarrow{(n_2,\varphi_2,d_2)} p_l \xrightarrow{(n_3,\varphi_3,d_3)} p_i$, then the boundedness criterion will be

$$a_z = a_x \frac{n_3}{d_3} + a_y \frac{d_2}{n_2}$$

Similar analysis can be performed on channels with two consumers and one producer. This construct represents the deterministic select operator.

Example 2.15. Figure 2.18.(a) represents an UCSDF graph with a simple channel e_1 and a multichannel $e_2 = (p_1, p_2, p_3, x, y, z)$. We would like to find a consistent EDF schedule with a totally ordered communication between p_1 and p_2 (i.e. no preemption). Since $\theta(e_1) = 2$ and $\delta(e_1) = 4$, all the possible affine relations between p_1 and p_2 that prevent overflow and underflow exceptions over channel e_1 are of the form $(4, \varphi_1, 6)$ such that $\varphi_1 \in \{0, 1\}$. Both the affine relations give the same order of firings of p_1 and p_2 which is $(p_1\ p_2\ p_1\ p_2\ p_1)^\omega$. Therefore p_1 and p_2 write into the multichannel in a deterministic manner (i.e. functional determinism is guaranteed). Let $s_1 = (1\ 0\ 1\ 0\ 1)^\omega$ (resp. $s_2 = \bar{s}_1 = (0\ 1\ 0\ 1\ 0)^\omega$) denote the instants at which p_1 (resp. p_2) fires. Channel e_2 can be splitted into two simple channels $e_{2,1}$ and $e_{2,2}$ whose consumer is actor p_3, as illustrated in Figure 2.18.(b). The consumption rate of $e_{2,1}$ is z_1; while the consumption rate of $e_{2,2}$ is z_2. So, $\forall j \in \mathbb{N}_{>0} : z(j) = z_1(j) + z_2(j)$. Let x^j be the total number of produced tokens by p_1 among the first j commonly produced tokens. So, x^j can easily be obtained from the sequence $(x \otimes s_1) + (y \otimes s_2)$. Hence, we have that $z_1(j) = x^{\oplus z(j)} - \oplus z_1(j-1)$. In this example, z_1 and z_2 are ultimately periodic sequences such that the length of their periods is equal to 18.

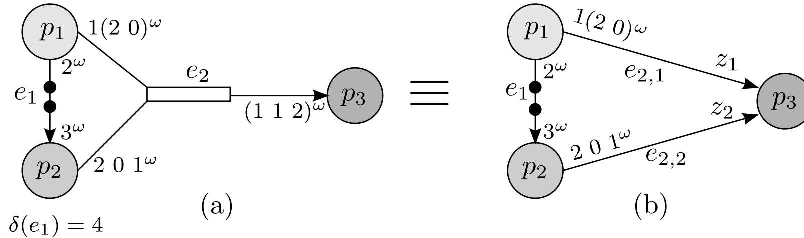


Figure 2.18: (a) An UCSDF graph with a multichannel and (b) its equivalent graph with simple channels.

Once we have a graph that consists only of simple channels, we can apply the affine relation synthesis described in Section 2.3. Nevertheless, we prefer to use the overflow and underflow analyses described in this section to handle multichannels because (1) the length of rate functions grows exponentially when we split multichannels as illustrated in the previous example, and (2) the computation of z_1 and z_2 depends on the number of initial tokens in the multichannel. So, if we would like to let $\theta(e_2)$ as a free variable, then z_1 and z_2 cannot be computed.

2.4.3 Shared storage space

So far, we have only supposed that channels are implemented as separated storage spaces. Let $G = (\{p_1, p_2\}, \{e_1 = (p_1, p_2, (0\,3\,0)^\omega, (3\,0\,0)^\omega), e_2 = (p_1, p_2, (0\,0\,3)^\omega, (0\,0\,3)^\omega)\})$ be a CSDF graph where $\theta(e_1) = \theta(e_2) = 0$. One possible affine relation between p_1 and p_2 that results in the minimum buffering requirements when using EDF scheduling is $(2, 3, 2)$. Figure 2.19 shows the number of accumulated tokens in e_1 and e_2 and the sum of them after each firing of p_1 and p_2 according to the affine relation and the scheduling policy. From that picture, we have that $\delta(e_1) = 3$ and $\delta(e_2) = 3$. So, when the channels

are implemented as separated buffers, the buffering requirements are equal to 6. But, we clearly notice that the sum of accumulated tokens in both channels at each instant of the execution is less or equal to 3. Hence, by implementing the two channels as a shared storage buffer, we can reduce the buffering requirements to 3.

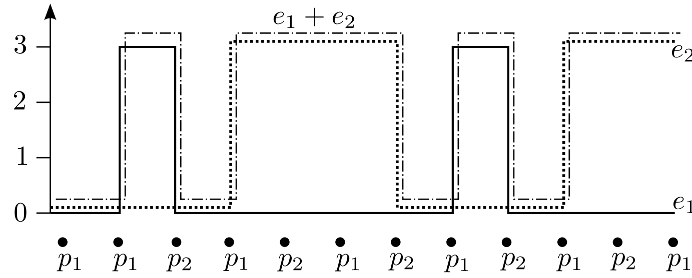


Figure 2.19: Buffer minimization using a shared storage space.

Let $e_1 = (p_1, p_2, x_1, y_1)$ and $e_2 = (p_3, p_4, x_2, y_2)$ be two channels in the dataflow graph. Without loss of generality, we suppose that they have the same data type. We would like to implement them as a single shared buffer b. The number of initial tokens in the shared buffer is $\theta(b) = \theta(e_1) + \theta(e_2)$; while its size is $\delta(b) \leq \delta(e_1) + \delta(e_2)$. Firstly, to ensure functional determinism, we need to devise a read and write mechanism so that each consumer pulls from the shared buffer only tokens written by its corresponding producer. One simple solution is to tag produced tokens by the ID of the producer.

The underflow analysis does not change; i.e. we must always have that

$$\forall j \in \mathbb{N}_{>0} : \theta(e_1) + \oplus x_1(\mathsf{cbef}_{2,1}(j)) - \oplus y_1(j) \geq 0$$
$$\forall j \in \mathbb{N}_{>0} : \theta(e_2) + \oplus x_2(\mathsf{cbef}_{4,3}(j)) - \oplus y_2(j) \geq 0$$

However, the overflow equations for the two channels must be combined together. Let $\mathsf{crbef}_{i,k} : \mathbb{N}_{>0} \longrightarrow \mathbb{N}$ be an integer function such that $p_k[\mathsf{crbef}_{i,k}(j)]$ is the last job of p_k that ***could*** write some of its results on channel e_2 before or at the same time at which $p_i[j]$ finishes writing its results on channel e_1. The overflow analysis can be written as

$$\forall j \in \mathbb{N}_{>0} : \theta(b) + \oplus x_1(j) + \oplus x_2(\mathsf{crbef}_{1,3}(j)) - \oplus y_1(\mathsf{cbef}_{1,2}(j)) - \oplus y_2(\mathsf{cbef}_{1,4}(j)) \leq \delta(b)$$

$$\forall j \in \mathbb{N}_{>0} : \theta(b) + \oplus x_1(\mathsf{crbef}_{3,1}(j)) + \oplus x_2(j) - \oplus y_1(\mathsf{cbef}_{3,2}(j)) - \oplus y_2(\mathsf{cbef}_{3,4}(j)) \leq \delta(b)$$

In the previous example, we have that $\mathsf{crbef}_{1,1}(j) = j$ and $\mathsf{cbef}_{1,2}(j) = \mathsf{prd}_{1,2}(j) = \max\{0, j-2\}$. Hence, the overflow equation is

$$\forall j \in \mathbb{N}_{>0} : \theta(b) + \oplus x_1(j) + \oplus x_2(j) - \oplus y_1(\mathsf{prd}_{1,2}(j)) - \oplus y_2(\mathsf{prd}_{1,2}(j)) \leq \delta(b)$$

2.4.4 FRStream

In the same spirit of Lucy-n, we propose a simple synchronous language called FRStream (FR from Firing Related) by expressing the synchronous semantics in our priority-driven semantics. In the synchronous paradigm, computations and communications

are instantaneous and scheduling precedences inside each reaction are deduced from the flow in the program. We can tailor the priority-driven semantics to express such paradigm as follows. Let $[s_{i,k}, s_{k,i}]$ be an activation relation between actors p_i and p_k. If $p_i[j] < p_k[j']$, then $p_i[j]$ executes entirely before $p_k[j']$ because an actor takes zero time to fire. This is similar to the totally ordered communication strategy in EDF scheduling policy. The main difference lies when $p_i[j]$ and $p_k[j']$ are synchronous. Suppose that there is a channel from p_i to p_k. In a synchronous language such as SIGNAL, there will be a scheduling precedence between $p_i[j]$ to $p_k[j']$ because $p_k[j']$ will use the value produced by $p_i[j]$ at the same instant (i.e. zero delay communication). In our case, it is possible that $p_k[j']$ will not use any value produced by $p_i[j]$ because of the buffering mechanism. Therefore, we add a scheduling precedence only if results of $p_i[j]$ are required by $p_k[j']$. Regarding auto-concurrency, if $p_i[j] = p_i[j']$ with $j' > j$, then we add a scheduling precedence between $p_i[j]$ and $p_i[j']$.

A Language constructs

The main constructs of our language are *streams*, *actors*, *activation relations*, and *processes*. Syntactically, an infinite sequence uv^ω is denoted by `u(v)*`; while a finite sequence uv^n is denoted by `u(v)n`.

1) Streams: Every variable in the program is a stream. So, `integer x init 0(1)2;` means that x is a stream of integers that initially contains values 0, 1, and 1. Let s be an ultimately periodic integer sequence such that $\forall j \in \mathbb{N}_{>0} : s(j) > 0$. During the jth firing of actor p, expression $x[s]$ at the right hand of an assignment denotes the $s(j)^{th}$ token consumed by $p[j]$ from stream x. Similarly, expression $x[s]$ at the left hand of an assignment denotes the $s(j)^{th}$ token produced by $p[j]$ on stream x. When $s = i^\omega$ with $i \in \mathbb{N}_{>0}$, we use notation $x[i]$ instead of $x[(i)*]$. At each firing, a value can be read many times but can be only defined once (i.e. single assignment).

2) Actors: An actor is a deterministic program such that each pair of its sub-actors is $(1, 0, 1)$-affine-related. All arithmetic and logic operators are actors. For instance, `plus(1 2,0 3)=1 5`. Listing 2.2 represents an actor p_1 that has an input stream x and output stream y. As in SIGNAL, symbol "|" denotes synchronous composition. So, the two sub-actors (the copy and plus actors) are synchronous and have a data-dependency (stream z). However, they can run in no particular order because z is never empty. Note that though there is no assignment to $y[1]$ the default value will be produced at each firing since we cannot produce the *second* token without producing the *first* one. Thus, both the production and consumption rates of p_1 are equal to 2^ω. Table 2.2 represents a synchronous execution of actor p_1. Listing 2.3 represents the code of an actor that has two states s_1 and s_2. A different firing function is executed in each state. When p_2 is in state s_1, it switches to state s_2 after one firing; and then it returns back to state s_1 after one firing. Only ultimately periodic state automata are allowed. The production rate of stream y is equal to $(0\ 2)^\omega$; while the consumption rate of stream x is equal to $(2\ 0)^\omega$.

Listing 2.2: Actor p_1.

```
actor P1(?real x; ! real y;)
  real z init 4.0;
begin
 state S1{
     y[2]=plus(x[2],z[1]) |
         z[1]=x[1]
 }
 transitions{ }
end
```

Listing 2.3: Actor p_2.

```
actor P1(?real x; ! real y;)
  real z;
begin
 state S1{  z[1]=x[1] | z[2]=x[2] }
 state S2{  y[2]=z[1] | y[1]=z[2] }
 transitions{
     S1 -- 1 ->> S2;
     S2 -- 1 ->> S1; }
end
```

Table 2.2: A synchronous execution of actor p_1.

firing	1	2	3
$x = 6\ 5\ 4\ 3\ 2\ 1$	4 3 2 1	2 1	ϵ
$z = 4$	6	4	2
$y = \epsilon$	0 9	0 9 0 9	0 9 0 9 0 5

3) Activation relations: In a process, every two *communicating* actors (we assume that instances of the same actor have different names) are $(1, 0, 1)$-affine-related by default. To express a different activation relation between two actors p and q, we use either notation `p,q=(n,phi,d);` to express an affine relation or `p,q=[s1,s2];` to express an ultimately periodic activation relation.

4) Processes: A process is a multi-clocked composition of actors and processes. Listing 2.4 is a process p that consists of the synchronous composition of two $[(1\ 0)^\omega, (0\ 2)^\omega]$-activation-related actors p_1 and p_2. Figure 2.20 represents a synchronous execution of process p.

Listing 2.4: Process p.

```
process P()
  real x init 0.0 1.0; real y;
(|
     y=P1(x) | x=P2(y)
|)
where P1,P2=[(1 0)*,(0 2)*];
end
```

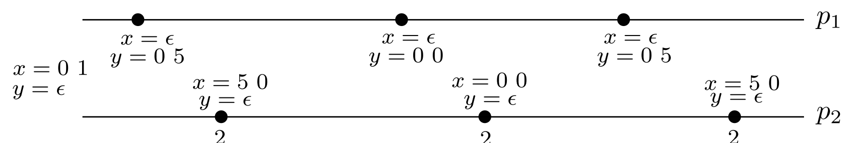


Figure 2.20: A synchronous execution of process p.

B Type system

In addition to classical type checking, the type system must ensure three other properties.

1) Consistency: The type system must statically check the consistency of the set of activation relations according to Propositions 2.5 and 2.6. This problem is decidable since all activation relations and rate functions are ultimately periodic.

2) Boundedness and deadlock freedom: We have two ensure that each stream is bounded and contains enough initial tokens. Each stream has one producer p_i and one consumer p_k. Firstly, we have to deduce the activation relation $[s_{i,k}, s_{k,i}]$, the production rate, and the consumption rate. Then, the overflow and underflow equations are applied such that functions $\mathsf{cbef}_{i,k}$ and $\mathsf{cbef}_{k,i}$ are computed according to the above described synchronous semantics.

3) Causality: This analysis is specific to the synchronous operational semantics. Consider the previous process (Listing 2.4) with an activation relation `P1,P2=[(1)*,(2)*]`. In this case, there are some synchronous activations; indeed $p_1[j] = p_2[2j-1] = p_2[2j]$. From the data-dependencies, $p_1[j]$ executes first if stream y is empty and $p_2[2j-1]$ executes first if stream x is empty. Therefore, there is a dependency cycle if both streams are empty. This problem is specific to the synchronous semantics because in EDF or FP scheduling, it is impossible to have for instance that $\mathsf{cbef}_{2,1}(\mathsf{cbef}_{1,2}(j)) = j$.

2.5 Conclusion

In this chapter, we have defined abstract schedules of dataflow graphs and presented the necessary conditions that a consistent schedule must satisfy. We have also refined the results for specific cases: strictly periodic schedules with EDF and FP scheduling policies, ultimately cyclo-static dataflow graphs, mulichannels, synchronous operational semantics, etc. In the next chapter, we will further refine abstract schedules by computing the necessary timing and scheduling parameters that maximize a performance metric and satisfy schedulability for different architectures and scheduling policies.

Chapter 3

Symbolic schedulability analysis

Contents

3.1 General conditions **87**
3.2 Fixed-priority scheduling **90**
3.2.1 Priority assignment 91
3.2.2 Uniprocessor scheduling 97
3.2.3 Multiprocessor scheduling 101
3.3 EDF scheduling **102**
3.3.1 Deadlines adjustment 102
3.3.2 Uniprocessor scheduling 106
3.3.3 Multiprocessor scheduling 109
3.4 Conclusion **112**

In real-time scheduling of dataflow graphs, each actor is mapped to a real-time periodic task. The necessary timing parameters that must be computed for each task are: periods, phases, and deadlines. More scheduling parameters could be needed as priorities of tasks in FP scheduling and processor allocation in partitioned multiprocessor scheduling. The computed scheduling parameters should respect the abstract schedule, ensure schedulability, and maximize a performance metric.

In the first section, we will present the general conditions that must be satisfied whatever the scheduling policy. Then, we present the symbolic schedulability analysis for two scheduling policies: FP scheduling (Section 3.2) and EDF scheduling (Section 3.3).

3.1 General conditions

Let $G = (P, E)$ be a timed static dataflow graph and G_r be the computed affine schedule (or more precisely the graph of affine relations). In a timed dataflow graph, each actor p_i is associated with a worst-case execution time sequence $\eta_i \in \mathbb{N}^\omega$ such that $\eta_i[j]$ is

the worst-case execution time of firing $p_i[j]$. To fit the periodic task model, we take the worst-case execution time of an actor p_i as $C_i = \max_j \eta_i[j]$. So, each actor p_i is characterized by (π_i, r_i, d_i, C_i) such that π_i is the period, r_i is the phase, and d_i is the relative deadline. In FP scheduling, actor p_i is permanently assigned the priority w_i. The dataflow is to be scheduled on a many-core processor architecture that consists of $M \geq 1$ identical cores. In partitioned scheduling, actor p_i is permanently allocated to core number ν_i.

An affine relation is an abstract constraint that describes the relative positioning of activations of two actors without considering the physical duration between two activations. In the sequel, we will impose that the time interval between successive activations of an actor is constant since an actor will be mapped to a *periodic* task. Let p_i and p_k be two (n, φ, d)-affine-related actors. We have hence that

$$d\pi_i = n\pi_k \tag{3.1}$$

$$r_k - r_i = \frac{\varphi}{n}\pi_i \tag{3.2}$$

So, the period and phase of p_k can be expressed in terms of the period and phase of p_i; i.e. $\pi_k = \frac{d}{n}\pi_i$ and $r_k = r_i + \frac{\varphi}{n}\pi_i$. But, timing parameters are non-negative integers. Thus, π_i must be a multiple of $\mathrm{lcm}\{\frac{n}{\gcd\{n,d\}}, \frac{n}{\gcd\{n,\varphi\}}\}$ and r_i must be greater or equal to $\frac{-\varphi}{n}\pi_i$. Similarly, if $p_i \xrightarrow{(n,\varphi,d)} p_k \xrightarrow{(n',\varphi',d')} p_l$, then $\pi_l = \frac{dd'}{nn'}\pi_i$ and $r_l = r_i + \frac{\varphi n' + \varphi' d}{nn'}\pi_i$.

• If the graph of affine relations is connected, then the period of each actor can be expressed in terms of the period of an arbitrary actor. We can therefore put $\forall p_i \in P : \pi_i = \alpha_i T$ where $\alpha_i \in \mathbb{Q}_{>0}$ and T is a multiple of some integer B. We can also put $\forall p_i \in P : r_i = f + \alpha'_i T$ such that f is the phase of the arbitrary actor (f must be greater than or equal to $\max_{p_i \in P}\{-\alpha'_i T\}$). Furthermore, we have that $U = \sum_{p_i \in P} \frac{C_i}{\pi_i} = \sum_{p_i \in P} \frac{C_i}{\alpha_i T} = \frac{\sigma}{T}$. Regardless of the scheduling policy, a necessary condition for schedulability is that $U \leq M$. Hence, we have that

$$T \geq \frac{\sigma}{M} \tag{3.3}$$

For some real-time systems, frequencies of tasks must be higher than some minimal frequencies under which safety is not ensured or the service quality is poor. The frequencies should also be lower than some maximal frequencies over which a device, for example, may get damage. To better meet these requirements, we allow the designer to specify upper bounds and lower bounds on periods of actors. We will then have the following condition.

$$T^l \leq T \leq T^u \tag{3.4}$$

where $T^l, T^u \in \mathbb{N}_{>0}$. The schedulability region concerning the periods and phases is initially defined by the following constraints: $T^l \leq T \leq T^u$, $T = kB$, and $T \geq \frac{\sigma}{M}$. So, T is the only free variable.

• If the graph of affine relations is disconnected, then we partition G_r to a set of $\mathcal{L}$ (maximal) connected graphs $G_j = (P_j, R_j)$. Hence, the period and phase of each actor

in a partition P_j can be expressed in terms of the period and phase of an arbitrary actor in that partition; i.e. $\forall p_i \in P_j : \pi_i = \alpha_i T_j$ and $r_i = f_j + \alpha'_i T_j$ where T_j must be a multiple of B_j, $T_j^l \leq T_j \leq T_j^u$, and $f_j \geq -\alpha'_i T_j$. Let U^j be the total contribution of actors in P_j to the processor utilization factor. So, $U^j = \sum_{p_i \in P_j} \frac{C_i}{\pi_i} = \frac{\sigma_j}{T_j}$. A necessary schedulability condition is therefore given as

$$\sum_{j=1}^{\mathcal{L}} \frac{\sigma_j}{T_j} \leq M \tag{3.5}$$

Regarding deadlines, we consider either implicit deadlines or constrained deadlines. The designer may like to impose deadlines. However, since periods are unknown, the designer has to specify deadlines in terms of periods; e.g. $d_1 = \frac{\pi_1}{2} - 3$. Therefore, the deadline of an actor p_i can be expressed in terms of the period of an arbitrary actor in the connected subgraph that contains p_i. So, $\forall p_i \in P_j : d_i = \beta_i T_j - \beta'_i$ where $\beta_i \in \mathbb{Q}_{>0}$ and $\beta'_i \in \mathbb{N}$. Since deadlines are integers, $\beta_i T_j$ must be an integer. Thus, factor B_j must be recomputed. We should also ensure that $C_i \leq d_i \leq \pi_i$. So,

$$C_i \leq \beta_i T_j - \beta'_i \leq \alpha_i T_j \tag{3.6}$$

Therefore, the bounds T_j^l and T_j^u should also be recomputed.

Regarding priority assignment and processor allocation, these scheduling parameters are required in the affine relation synthesis step and buffer sizes computation. But, in order to compute them, we need to perform a symbolic schedulability analysis that requires performing affine scheduling. We break the cycle in the scheduling approach as follows. Priorities and processor allocation are needed in affine scheduling only to compute parameters φ of affine relations since parameters n and d can be obtained from the boundedness criterion. Parameters φ are needed in the symbolic schedulability analysis only to compute phases of actors. But, most schedulability tests consider the worst-case arrival scenario when all actors arrive at the same time. Thus, we do not need parameters φ to perform the symbolic schedulability analysis. This slightly influences the computation of factors B_j.

Most real-life applications are modeled as connected dataflow graphs. Therefore, we will focus more on the symbolic schedulability analysis of connected graphs. Obviously, symbolic schedulability analysis of disconnected graphs is more complicated.

Performance metrics

The design process of any embedded real-time system is driven by the objective of providing the best possible performance within the timing and resource constraints. Thus, we need to express the performance as a function of the design parameters (free variables) and then exploring the schedulability region (i.e. the region of the parameter space that corresponds to feasible designs) to find the best solution. So, if X is the vector of design parameters, $\mathbb{S}$ is the schedulability region, and F is the cost (reward or

utility) function, then the symbolic schedulability analysis problem can be formulated as

$$\begin{aligned} &\max F(X) \\ &\text{Subject to } X \in \mathbb{S} \end{aligned} \tag{3.7}$$

Many practical cost functions exist such as energy consumption, throughput, latency, etc. Some of them could be in conflict with each other as the throughput and buffering requirements of dataflow graphs. In the rest of this section, we will present some cost functions and the controlled parameters for each function. However, we will not consider metrics that depend on the phases of actors such as the end-to-end latency.

Buffering requirements: As shown in the previous chapter, some scheduling parameters influence the buffer sizes computation. In partitioned scheduling, allocating two actors to the same processor results in a smaller total sum of capacities of the channels between them. In FP scheduling, switching priorities of two actors may improve sizes of buffers between them. In EDF scheduling, adjusting deadlines (to implement the totally ordered communication strategy) may also reduce the buffering requirements.

Throughput: The throughput of an actor p_i is the average number of firings of p_i per unit of time and hence equals to its frequency $\frac{1}{\pi_i}$. If the graph of affine relations is connected, then the throughput of the graph is equal to $\frac{1}{\pi_i \vec{r}(i)}$. The repetition vector $\vec{r}$ can be easily obtained from parameters n and d of affine relations. Since $\pi_i = \alpha_i T$ and $U = \frac{\sigma}{T}$, maximizing the throughput is equivalent to maximizing the processor utilization factor and so to minimizing T.

If the graph of affine relations is disconnected (and so the dataflow graph is also disconnected), the throughput of each connected subgraph G_j can always be expressed in terms of T_j. However, it is not clear how to define the throughput of the whole graph. We hence choose, as in the connected case, to maximize the processor utilization factor.

If $\mathcal{T} = [T_1, \ldots, T_{\mathcal{L}}]$ (a lightweight way to denote a vector), then we may consider any cost function $F(\mathcal{T})$. For most practical metrics, it is expected that the performance of the system improves when tasks' rates are increased. Therefore and as in [38], we my consider cost functions such that $\nabla F \leq 0$ (i.e. $\forall j : \frac{\partial F}{\partial T_j} \leq 0$). For example, if $F(\mathcal{T}) = U = \sum\limits_{j=1}^{\mathcal{L}} \frac{\sigma_j}{T_j}$, then $\frac{\partial F}{\partial T_j} = \frac{-\sigma_j}{T_j^2} \leq 0$. Another example is the first order approximation of the energy spent in a bubble control application which is given as $\sum\limits_{p_i \in P} a_i e^{-\frac{b_i}{\pi_i}}$ [164].

3.2 Fixed-priority scheduling

In this section, we present the symbolic FP schedulability analysis for uniprocessor and multiprocessor systems. In each case, we describe the solution with respect to buffer minimization or processor utilization maximization.

3.2.1 Priority assignment

We will describe here how to compute the priority of each actor (without computing the timing parameters) w.r.t. buffer minimization, processor utilization maximization, or a combination of the two metrics. Priorities are used in affine scheduling and they will be used later to perform a symbolic response time analysis.

A Buffer minimization

Let $E_{i,k} \subseteq E$ be the set of all channels between p_i and p_k (in both directions); and let $b_{i,k}$ be the total sum of approximate capacities of channels in $E_{i,k}$ assuming that p_i has a higher priority than p_k (i.e. $w_i < w_k$). Table 2.1 (p. 76) shows how to compute the approximate size of a channel (column $\nu_i = \nu_k$). So, $b_{i,k} = \sum_{e \in E_{i,k}} F(e)\delta(e)$ such that $w_i < w_k$. For each connected subgraph $G_j = (P_j, E_j)$ in the graph of affine relations, we construct a weighted complete directed graph $G'_j = (P_j, E'_j)$ such that the weight of edge $e_{i,k} = (p_i, p_k) \in E'_j$ is equal to $b_{i,k}$.

Hence, the problem of finding the priority assignment that reduces the buffering requirements is equivalent to the linear ordering problem (LOP) [135] which consists in finding an *acyclic* tournament in the graph G'_j such that the sum of weights of edges in the tournament is minimal. A tournament is a subset of edges containing for every pair of nodes p_i and p_k either edge (p_i, p_k) or (p_k, p_i) but not both. For N actors, there are $N!$ different priority assignments (i.e. permutations). The LOP is NP-hard; however many exact and heuristic solutions had been proposed in the past decades [135].

The LOP can be formulated as a 0/1 integer linear program. A 0/1 variable $x_{i,k}$ states whether edge (p_i, p_k) is present in the tournament or not.

$$
\begin{aligned}
&\min \sum_{i,k} b_{i,k}\ x_{i,k} \\
&\forall i < k : x_{i,k} + x_{k,i} = 1 \\
&\forall i < j, i < k, j \neq k : x_{i,j} + x_{j,k} + x_{k,i} \leq 2 \\
&\forall i, k : x_{i,k} \in \{0, 1\}
\end{aligned}
$$

Once the LOP is solved for every connected subgraph, actors within each subgraph can be ordered according to the computed priorities. It does not matter how to order two actors in different subgraphs because this will not influence the buffering requirements.

Example 3.1. Consider the UCSDF example in Figure 2.14 (p. 79) and assume that all tokens have the same size. The affine relations in the graph are $p_1 \xrightarrow{(2,\varphi_1,4)} p_2 \xrightarrow{(6,\varphi_2,2)} p_3 \xrightarrow{(4,\varphi_3,6)} p_1$. The graph of affine relations is connected and the constructed weighted graph for LOP assignment is depicted in Figure 3.1. So, the best priority ordering is $p_2\ p_1\ p_3$; while the worst priority ordering is $p_3\ p_1\ p_2$ (which is the RM priority assignment).

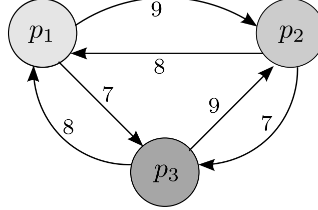


Figure 3.1: Priority assignment problem expressed as a LOP.

B Utilization maximization

If deadlines are free parameters, then using implicit deadlines obviously results in the best utilization. Let us consider the case where deadlines are implicit or user-imposed. In both cases, we have that $\forall p_i \in P_j : d_i = \beta_i T_j - \beta'_i$. As mentioned in Chapter 1, the DM priority assignment is optimal (for synchronous task sets). Therefore, it results in the largest processor utilization. In case of equal deadlines, we break the tie using the actors' ID. So, the (initial) schedulability region (or the $\mathcal{T}$-space) can be partitioned into a set of priority regions as illustrated by the following example.

Example 3.2. We add to the UCSDF example (Figure 2.14, p. 79) two actors p_4 and p_5 and a channel $e_4 = (p_4, p_5, 4^\omega, 1^\omega)$. We have hence an additional affine relation $p_4 \xrightarrow{(8,\varphi_4,2)} p_5$ in graph G_r which is disconnected in this case. So, we have that $\pi = [T_1, 2T_1, \frac{2}{3}T_1, T_2, \frac{T_2}{4}]$. Suppose that $C = [65, 70, 95, 60, 55]$ and $d = [\pi_1 - 30, \frac{\pi_2}{2}, \frac{3}{4}\pi_3 - 10, \frac{\pi_4}{5}, \pi_5 - 60]$. We have that $U = \frac{242.5}{T_1} + \frac{280}{T_2}$. The initial $\mathcal{T}$-space is hence defined by $T_1 \geq 210, T_2 \geq 460, B_1 = 6,\ B_2 = 20$, and $U \leq 1$ (i.e. uniprocessor scheduling). Using linear equations $d_i = d_k$, we can partition the $\mathcal{T}$-space into a set of priority regions as illustrated in Figure 3.2. For instance region $\mathbb{S}_1$ corresponds to the DM priority ordering $p_3\ p_5\ p_4\ p_1\ p_2$.

Definition 3.1 (Monotonicity of schedulability)**.** FP schedulability is monotone over the $\mathcal{T}$-space if and only if given any two points $\mathcal{T}_1$ and $\mathcal{T}_2$ in the $\mathcal{T}$-space such that $\mathcal{T}_1 \leq \mathcal{T}_2$ and the task set is FP schedulable at $\mathcal{T}_1$, then the task set is also FP schedulable at $\mathcal{T}_2$.

Property 3.1. Fixed-priority schedulability with *predefined* priorities is monotone over the $\mathcal{T}$-space.

Proof: We mean by "predefined" priorities the precomputed priorities that do not depend on the values of T_j (e.g. the priorities computed for buffer minimization using LOP assignment). To prove Property 3.1, we use the response time analysis. We have that $R_i = C_i + \sum_{w_i < w_k} \left\lceil \frac{R_i}{\pi_k} \right\rceil C_k$. When periods are increased, priorities do not change; hence, the solution of the recurrence (i.e. the worst-case response time) decreases. Furthermore, the deadline d_i gets larger since $d_i = \beta_i T_j - \beta'_i$. So, if R_i was initially less than or equal to d_i, then this remains true when periods are increased. □

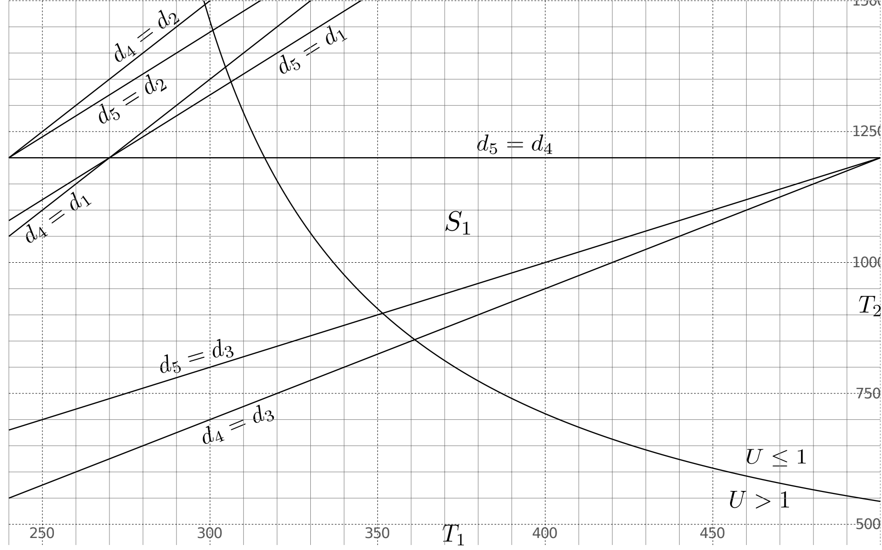


Figure 3.2: Partitioning of the $\mathcal{T}$-space into a set of DM priority regions.

Proposition 3.1. DM schedulability is monotone over the $\mathcal{T}$-space.

Proof: Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two points in the $\mathcal{T}$-space such that $\mathcal{T}_1 \leq \mathcal{T}_2$ and the task set is DM schedulable at point $\mathcal{T}_1$. We denote by π^i and d^i the vectors of periods and deadlines at point $\mathcal{T}_i$, respectively. We have that $\pi^1 \leq \pi^2$ and $d^1 \leq d^2$. There are two cases:
1. If $\mathcal{T}_1$ and $\mathcal{T}_2$ are in the same priority region, then the task set is also DM schedulable at point $\mathcal{T}_2$ according to Property 3.1.
2. If $\mathcal{T}_1$ and $\mathcal{T}_2$ are in different priority regions, then we break the transformation from the first to the second point into successive swaps as illustrated by the following example.

Let $\pi^1 = (12, 30, 25, 24)$, $d^1 = (5, 20, 21, 24)$, $\pi^2 = (35, 32, 25, 25)$, and $d^2 = (27, 26, 21, 25)$. So, we have that $d^1 \leq d^2$ and $\pi^1 \leq \pi^2$. But, $\mathcal{T}_1$ and $\mathcal{T}_2$ are in different priority regions since the DM priority ordering at $\mathcal{T}_1$ is p_1 p_2 p_3 p_4 while the DM priority ordering at point $\mathcal{T}_2$ is p_3 p_4 p_2 p_1. To get from the first permutation (or ordering) to the second one, we need five swaps (see Figure 3.3).

Since the task set is DM schedulable at the first point, we have only to prove that it remains DM schedulable after each swap. So, at each step, we switch only priorities of p_i and p_{i+1} (assuming that actors are ordered by their priorities). According to the worst-response time computation, we have that

- $\forall k > i+1 : R_k$ remains intact (Actually, R_k decreases if periods π_j s.t. $j < k$ are increased). Hence, $\forall k > i+1 : p_k$ meets its deadline.

- $\forall k < i$ or $k = i+1$: R_k decreases or remains intact (note that p_i is excluded from the sum in the response time formula).

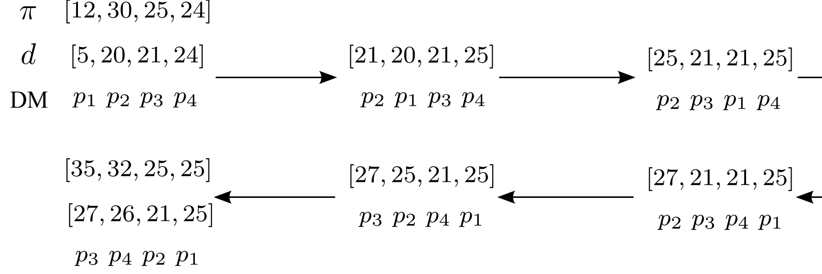


Figure 3.3: Transforming DM priority assignments.

- For p_i, we have that $R_i = C_i + \left\lceil \frac{R_i}{\pi_{i+1}} \right\rceil C_{i+1} + \sum_{k<i} \left\lceil \frac{R_i}{\pi_k} \right\rceil C_k$. Let R'_{i+1} and d'_{i+1} be the response time and deadline, respectively, of actor p_{i+1} before the swap. So, we have that $R'_{i+1} = C_{i+1} + \left\lceil \frac{R'_{i+1}}{\pi_i} \right\rceil C_i + \sum_{k<i} \left\lceil \frac{R'_{i+1}}{\pi_k} \right\rceil C_k$. If we can prove that $R_i \le R'_{i+1}$, then we will have that $R_i \le d_i$ because $d_i \ge d'_{i+1} \ge R'_{i+1}$.

Since R'_{i+1} is the fixed point of the response time formula, we have that $R'_{i+1} = C_{i+1} + a_i C_i + \sum_{k<i} a_k C_k$ such that $\forall k \le i : a_k = \left\lceil \frac{R'_{i+1}}{\pi_k} \right\rceil \in \mathbb{N}_{>0}$. Let us solve the recurrence for R_i starting from $R_i^0 = R'_{i+1}$. So, $R_i^1 = C_i + \left\lceil \frac{R'_{i+1}}{\pi_{i+1}} \right\rceil C_{i+1} + \sum_{k<i} a_k C_k$. But, $R'_{i+1} \le \pi_{i+1}$ because the system is DM schedulable before the swap. Thus, $R_i^1 = C_i + C_{i+1} + \sum_{k<i} a_k C_k \le R'_{i+1}$. We have two cases, either $R_i^1 = R'_{i+1}$ (the recurrence stops) or $R_i^1 < R'_{i+1}$. In the second case, we will continue until reaching a fixed point which will be less than R'_{i+1}. □

One direct result of Proposition 3.1 is that if a task is not DM schedulable at point $\mathcal{T}_2$, then it is not DM schedulable at point $\mathcal{T}_1$.

C Buffering vs processor utilization

As in static-periodic scheduling of (C)SDF graphs [180], buffer minimization and throughput maximization are in conflict with each other. Indeed, we have shown in Example 3.1 that the worst buffering requirements are obtained by the DM priority ordering. This observation can be simply explained as follows. Let p_i and p_k be two (n, φ, d)-affine-related actors with implicit deadlines and all channels between them are directed from p_i to p_k. We have that $\pi_k = \frac{d}{n}\pi_i$. If $d > n$, then p_i will have a higher DM priority than p_k. But according to Table 2.1 (p. 76), p_k must have a higher priority than p_i in order to achieve better buffer sizes.

Let O_1 and O_2 be two priority orderings (i.e. permutations). The *swapping distance* between O_1 and O_2 is the minimum number of transpositions of two adjacent elements necessary to transform permutation O_1 into permutation O_2. Each step in the swapping path positively or negatively affects the buffering requirements and the throughput. The impact on buffer sizes can be approximately measured by equations in Table 2.1.

Example 3.3. Let us consider the dataflow graph in Figure 2.14 (p. 79). We add two actors p_4 and p_5 and two channels $e_4 = (p_4, p_5, 4^\omega, 1^\omega)$ and $e_5 = (p_5, p_3, 1^\omega, 2^\omega)$. So, $\pi = [T, 2T, \frac{2}{3}T, \frac{4}{3}T, \frac{T}{3}]$, $C = [65, 70, 95, 60, 55]$ and $U = \frac{452.5}{T}$. We take deadlines as $d = [\pi_1 - 30, \frac{\pi_2}{2}, \frac{3}{4}\pi_3 - 10, \frac{\pi_4}{2}, \pi_5 - 60]$. So, the initial $\mathcal{T}$-space is defined by $B = 6$ and $T \geq 452.5$. Figure 3.4 represents the trade-off between processor utilization and approximate buffering requirements for all the possible (5!) priority orderings. As one could notice, many priority orderings can result in the same buffering requirements but in different processor utilizations. If we consider only Pareto points, then the buffering requirements and the throughput are proportional to each other.

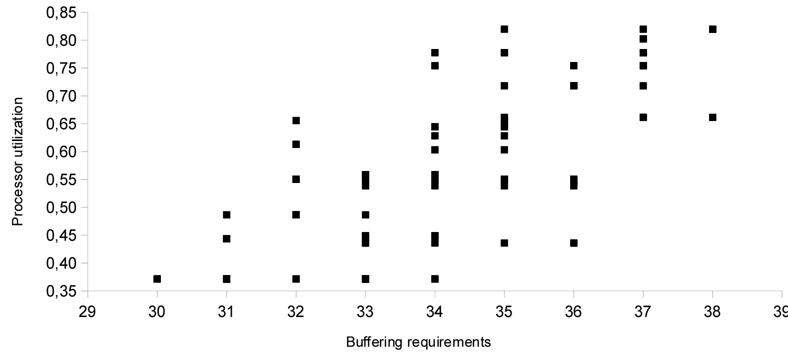


Figure 3.4: Exploration of the priority orderings space: throughput vs buffering requirements.

It will be useful if one can compute the (approximate) cost of a swap on the processor utilization without performing a symbolic schedulability analysis. Let O^*, π^*, U^* be the DM priority ordering, the period vector, and the processor utilization, respectively, obtained by the symbolic schedulability analysis. Hence, any priority ordering should result in a processor utilization no greater than U^*. Figure 3.5 shows for every priority ordering O the precedence distance between O and O^* and the corresponding decrease in processor utilization. The precedence distance between two permutations O and O^* is defined as follows. Let $\text{sim}(O, O^*)$ be equal to the number of times an actor p_i is preceded by an actor p_k in both O and O^*. The precedence distance is hence equal to

$$\frac{N(N-1)}{2} - \text{sim}(O, O^*)$$

If $\tilde{x}_{i,k}$ is the 0/1 variable that states whether actor p_i has a higher priority in O^* than p_k or not, then the precedence distance is equal to $\frac{N(N-1)}{2} - \sum_{i,k} x_{i,k}\, \tilde{x}_{i,k}$. In the ILP formulation, we could add a constraint that states that the precedence distance is less than some threshold. From Figure 3.5, the decrease in the processor utilization is ***roughly*** proportional to the precedence distance. However, the precedence metric is

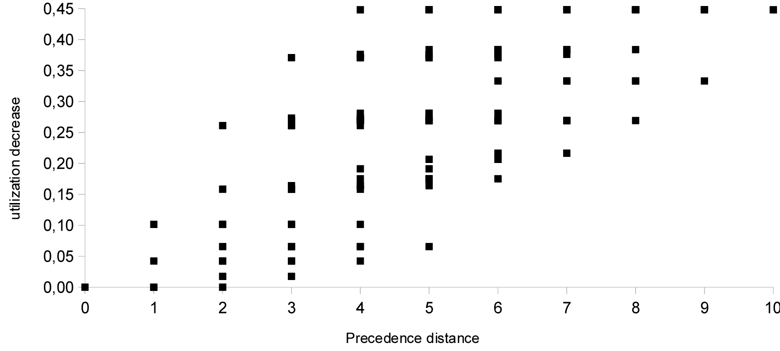


Figure 3.5: Exploration of the priority orderings space: throughput vs precedence distance.

not accurate and it is possible to define a better one. Figure 3.6 presents a new more accurate distance metric called the utilization distance.

Utilization distance: We compute a new period vector π for which the task set is schedulable w.r.t. priority ordering O. Vector π must be computed without performing a complex schedulability analysis. If U is the processor utilization that corresponds to period vector π, then the utilization distance is equal to

$$0 \leq \text{dis}(O, O^*) = 1 - \frac{U}{U^*} \leq 1$$

Vector π is computed as follows. We compute the response times w.r.t. periods π^* and priority ordering O. Then, we take $T_j = \max\{T_j^*, \max_{p_i \in P_j}\{\left\lceil \frac{R_i + \beta_i'}{B_j \beta_i} \right\rceil B_j\}\}$. Clearly, the task set is FP schedulable w.r.t. the new computed periods.

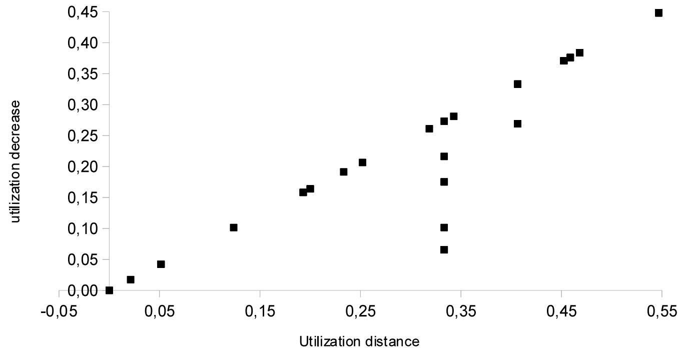


Figure 3.6: Exploration of the priority orderings space: throughput vs utilization distance.

Heuristic: Many heuristics have beed developed to solve the linear order problem [135]. We choose a simple local enumeration method. For a given ordering and a given window of length W, the heuristic (Algorithm 3) arranges the elements in the window to get the best buffering requirements and an utilization distance no greater than a threshold u. We denote by $b(O)$ the sum of approximate buffer sizes w.r.t. priority ordering O. The best ordering among a set of orderings is the one with the smallest buffering requirements. In case of tie, we choose the one with the smallest utilization distance.

Algorithm 3: Priority ordering heuristic

```
O = O*;
repeat
    O_prev = O;
    for i=1, ..., N-W+1 do
        Find the best priority ordering O' s.t. dis(O', O*) ≤ u by arranging
        elements at positions i, i+1, ..., i+W-1 in O;
        if b(O') < b(O) or ( b(O') = b(O) and dis(O', O*) < dis(O, O*) ) then
            O = O';
until O_prev == O;
```

3.2.2 Uniprocessor scheduling

We firstly consider the simple case where the graph of affine relations is connected and there is only one priority region (e.g. when deadlines are implicit, the LOP assignment, etc). The maximum throughput corresponds to the minimum value of T. For RM scheduling, we can use the utilization-based tests but they are too pessimistic. Therefore, we will use a symbolic RTA. In the first place, we reduce the space of feasible values of T by using upper and lower bounds on worst-case response times. As shown in Chapter 1,

$$R_i^u = \frac{C_i + \sum_{w_k < w_i} C_k(1 - U_k)}{1 - \sum_{w_k < w_i} U_k}$$

Following a similar reasoning to that in [40], we can prove that if the system is schedulable, then

$$R_i^l = \frac{C_i - \sum_{w_k < w_i} C_k(1 - U_k)}{1 - \sum_{w_k < w_i} U_k} \tag{3.8}$$

If $\exists p_i \in P : R_i^l > d_i$, then the task system is unschedulable. We have that

$$R_i^l > d_i \Leftrightarrow$$

$$\beta_i T^2 - (C_i + \beta_i' + \sum_{w_k < w_i} C_k(\frac{\beta_i}{\alpha_k} - 1))T - \sum_{w_k < w_i} \frac{C_k}{\alpha_k}(C_k - \beta_i') < 0$$

The second degree inequality can be easily solved to find an interval in which the system is unschedulable. Dually, if $\forall p_i \in P : R_i^u \leq d_i$, then the task set is schedulable. We have that

$$R_i^u \leq d_i \Leftrightarrow$$

$$\beta_i T^2 - (C_i + \beta_i' + \sum_{w_k < w_i} C_k (1 + \frac{\beta_i}{\alpha_k}))T + \sum_{w_k < w_i} \frac{C_k}{\alpha_k}(C_k + \beta_i') \geq 0$$

So, we have intervals in which the system is schedulable and intervals in which the system is unschedulable. To find the best T, it is sufficient to perform a dichotomic search in the remaining space knowing that if $R_i \leq d_i$ for some T, then $R_i \leq d_i$ for all $T' > T$. We denote this procedure by `SRTA`.

Example 3.4. Let us take the task set in Example 3.3 and the priority assignment $p_1\ p_2\ p_3\ p_4\ p_5$. The solutions of the previous inequalities are presented in Table 3.1. So, the system is schedulable for $T \in [1422, \infty[$ and unschedulable for $T \in [0, 456[$. To find the best T, we perform RTA for values in $[456, 1416]$ using a dichotomic search. Each time, we compute the response times in the order $p_5\ p_4\ p_3\ p_2\ p_1$; i.e. the task that gives the worst value of T s.t. $R_i^u \leq d_i$ is considered first. We will find the best value of T after computing only six response times. Clearly, the symbolic RTA is faster for large values of B.

Table 3.1: Symbolic RTA of the task set in Example 3.3 with DM priorities.

actor	$R_i^l > d_i$	$R_i^u \leq d_i$
p_1	$\emptyset$	$[456, \infty[$
p_2	$\emptyset$	$[456, \infty[$
p_3	$\emptyset$	$[558, \infty[$
p_4	$\emptyset$	$[630, \infty[$
p_5	$\emptyset$	$[1422, \infty[$

The second case is DM scheduling of connected graphs. Each DM priority region corresponds to an interval. There are at most $1 + \frac{N(N-1)}{2}$ DM priority regions since each equation $d_i < d_k$ can create only one new region. We use RTA to find the first interval $[t^l, t^u]$ such that the task set is schedulable for $T = t^u$. According to Proposition 3.1, the task set is schedulable for every $T > t^u$ and unschedulable for every $T < t^l$. To find the best T, we only have to apply the `SRTA` procedure in interval $[t^l, t^u]$.

The third case is when the graph of affine relations is disconnected and there is only one priority region. Since priorities are fixed a priori and the cost function $U(\mathcal{T})$ is convex, we can apply the branch-and-bound method described in [38] in case deadlines are equal to periods. We present here a different depth-first brand-and-bound (DF-B&B) solution based on symbolic RTA. However, the presented algorithm does not seek for a global optimum solution but for the optimum solution in the sub-space dominated

by a first admissible solution $\mathcal{T}^{\text{fst}}$ (i.e. for all $\mathcal{T}$ in the sub-space, $\mathcal{T} \leq \mathcal{T}^{\text{fst}}$). To get the global optimum, we have to take $\mathcal{T}^{\text{fst}} = \mathcal{T}^{u}$. Algorithm 4 represents the pseudo-code of the search algorithm.

Algorithm 4: DF-B&B SRTA.

```
Procedure main() begin
    T^cur = T^fst ;
    U^cur = U(T^fst);
    VisitTree(1);
    return T^cur;
Procedure visitTree(i) begin
    ⅂_i =PossibleValuesOf(T_i) ;
    for each t ∈ ⅂_i do
        if not Feasible(T^node) then
            prune this node and nodes for remaining values in ⅂_i
        else
            if U(T^node) < U^cur then
                T^cur = T^node ;
                U^cur = U(T^cur);
            if i < L then VisitTree(i+1);
```

In order to explain the algorithm more clearly, we solve the problem in Example 3.2 w.r.t. priority ordering p_4 p_2 p_3 p_1 p_5. Since there are two weakly connected components, each point in the $\mathcal{T}$-space has two components; i.e. $\mathcal{T} = [T_1, T_2]$. The start point of the algorithm is a first admissible solution $\mathcal{T}^{\text{fst}}$ (or simply $\mathcal{T}^{u}$, assuming that the task is schedulable at point $\mathcal{T}^{u}$).

We may compute $\mathcal{T}^{\text{fst}}$ using a linear relaxation of RTA as follows. Firstly, each deadline $d_i = \beta_i T_j - \beta'_i$ is under-approximated as $d'_i = \rho_i T_j$. So, we could simply take $\rho_i = \beta_i - \frac{\beta'_i}{T^l_j}$. Clearly, if the system is schedulable with the new deadlines, then it is schedulable with the original ones. Furthermore, the system is schedulable if $\forall p_i : R^u_i = \frac{C_i + \sum_{w_k < w_i} C_k}{1 - \sum_{w_k < w_i} U_k} \leq d'_i$. We solve this problem in $\mathbb{R}_{>0}$ as a linear program by putting $x_j = \frac{1}{T_j}$ while the objective function is to maximize the processor utilization. If $\mathcal{X} = [x_1, x_2, \ldots]$ is a solution of the linear program, then we take $T^{\text{fst}}_j = \left\lceil \frac{1}{x_j\ B_j} \right\rceil B_j$. Linear programming has a lower time complexity than convex optimization used in [38]. Regarding the previous example, we have that $d' = [\frac{6}{7}T_1,\ T_1, \frac{19}{42}T_1, \frac{1}{5}T_2, \frac{11}{92}T_2]$. The linear program can hence be written as

$$
\begin{aligned}
&\max 242.5\ x_1 + 280\ x_2 \\
&242.5\ x_1 + 280\ x_2 \le 1 && (U \le 1) \\
&300\ x_2 \le 1 && (R_4^u \le d_4') \\
&130\ x_1 + 60\ x_2 \le 1 && (R_2^u \le d_2') \\
&10115\ x_1 + 1140\ x_2 \le 19 && (R_3^u \le d_3') \\
&3095\ x_1 + 360\ x_2 \le 6 && (R_1^u \le d_1') \\
&2667.5\ x_1 + 32400\ x_2 \le 11 && (R_5^u \le d_5') \\
&0 \le x_1 \le \frac{1}{210}; \qquad 0 \le x_2 \le \frac{1}{460}
\end{aligned}
$$

The solution of this program is $\mathcal{T}^{fst} = [540, 5360]$ and $U^{\text{fst}} = 0.501$. So, the $\mathcal{T}$-space consists of 13776 points to be checked. From this first admissible solution, the tree structure built by the invocation of `visitTree` is shown in Figure 3.7. At the first level, procedure `PossibleValuesOf`(T_1) returns all possible values of T_1 (denoted by $\daleth_1$) in $[T_1^l, T_1^{\text{fst}}]$ such that the system could be schedulable at each point $\mathcal{T} = [t \in \daleth_1, T_2, \ldots]$; i.e. the values of the other components in vector $\mathcal{T}$ are constant. Values of $\daleth_i$ are obtained by using test $\forall p_i : R_i^l = \frac{C_i - \sum_{w_k < w_i} C_k(1-U_k)}{1 - \sum_{w_k < w_i} U_k} \le d_i$ since the system is unschedulable if $\exists p_i : R_i^l > d_i$. For the previous example, we have that $\daleth_1 = [258, 540]$. The nodes at this level are processed in the decreasing order of T_1. If the system is unschedulable at a given node, then the subtree whose root is that node will be pruned since, according to Property 3.1, reducing values of the other components (i.e. nodes in the subtree) will not make the system schedulable. For the same raison, we have also to prune all the nodes for the remaining values of T_1 in $\daleth_i$. In the example, the system is unschedulable for $T_1 = 468$ and so for all $T_1 \in [258, 468]$.

At level $\mathcal{L}$, there is only one free variable, we can hence use `SRTA` instead of the enumeration to compute the best value of the last component $T_{\mathcal{L}}$.

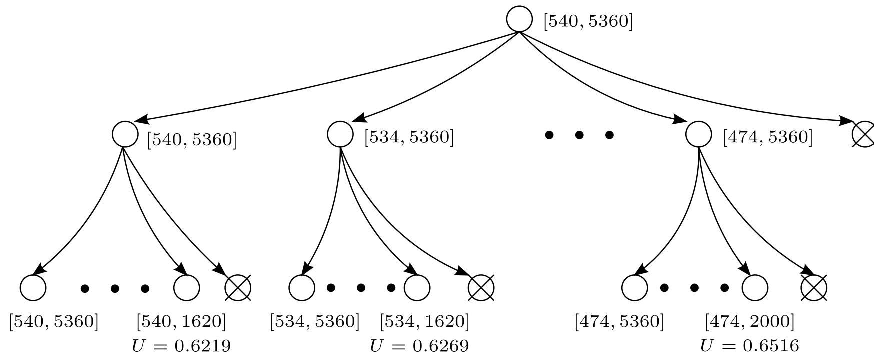


Figure 3.7: Illustration of DF-B&B symbolic FP schedulability analysis.

The fourth and last case is DM scheduling of disconnected graphs. We use the previous branch and bound technique but we have also to consider the DM priority regions. The first admissible solution can be obtained as in the previous case by considering any priority ordering. Hence, $\mathcal{T}^{\text{fst}} = [474, 2000]$ (the best solution for priority assignment $p_4\, p_2\, p_3\, p_1\, p_5$) is an admissible solution for the DM scheduling. Starting from this point, the space $210 \leq T_1 \leq 474 \wedge 460 \leq T_2 \leq 2000 \wedge U \leq 1$ contains 9 DM priority regions (see Figure 3.2). The DF-B&B algorithm can simply be applied on each region. However, this algorithm can be further improved; for instance, by considering Proposition 3.1. For example, Let $\mathcal{T}$ be the current node in the current DM priority region, and let $\mathcal{T}'$ be a node in an already processed region. If $\mathcal{T}' \geq T$ and the system is unschedulable in $\mathcal{T}'$, then the current node can be immediately pruned.

3.2.3 Multiprocessor scheduling

Let us first address the symbolic partitioned scheduling problem. We consider only connected dataflow graphs. Partitioned scheduling is known to be NP-hard; we use therefore a best fit heuristic that consists in two steps:

1. Order actors by their decreasing priorities. We will consider only systems with a unique priority region (e.g. computed by the LOP priority assignment).
2. Assign each actor (in the previous order) to a processor according to the best fit strategy.

Let $(V_i)_{i=1,M}$ be the initially empty M partitions (i.e. each partition corresponds to a processor). We allocate task p_k, assuming that higher priority tasks are already allocated, as follows. Let $\mathcal{T}_i$ be the optimal value (a point in the $\mathcal{T}$-space) returned by the symbolic response time analysis such that all partitions $(V_j)_{j\neq i}$ and $V_i \cup \{p_k\}$ are schedulable each one on a single processor. $\mathcal{T}_i$ can be found by applying `SRTA` only on partition $V_i \cup \{p_k\}$ (since partitions $(V_j)_{j\neq i}$ remain schedulable). We assign task p_k to partition V_i that results in the maximum processor utilization $U(\mathcal{T}_i)$. In case of equality, we break the tie by favoring the partition with the minimum utilization $U^i = \sum\limits_{p_j \in V_i \cup \{p_k\}} \frac{C_j}{\pi_j}$ and hence seek for balanced partitions.

For N periodic tasks, we will apply the symbolic response time analysis $N \times M$ times. However, adding task p_k to a partition V_i does not change the worst-case response times of the already allocated tasks on V_i. Hence, if T' is the minimum value for which V_i is schedulable (which is computed in the previous steps), then $T = \max\{T', \min\{T |\ R_k \leq d_k\}\}$ is the minimum value for which $\{p_k\} \cup V_i$ is schedulable. If we would like to speed up more the allocation procedure, we may take $T = \max\{T', \min\{T |\ R_k^u \leq d_k\}\}$ as an approximate schedulability test.

Example 3.5. Let us consider the task set in Example 3.3 with priority assignment $p_1\ p_2\ p_5\ p_4\ p_3$ and $M = 2$. Initially, we have that $V_1 = \emptyset$ and $V_2 = \emptyset$. Task p_1 is assigned directly to partition V_1 and task p_2 to partition V_2; so, $V_1 = \{p_1\}, V_2 = \{p_2\}$. At this stage, we have $T = 96$ and $U(T) = \frac{65}{96} + \frac{70}{192} = 1.041$. When allocating task

p_5, we have two cases, either to assign the task to partition V_1 or to partition V_2. In the first case, p_1 is already schedulable (for $T \geq 96$). We have that $T \geq 96$ and $R_5 = 55 + \lceil \frac{R_5}{T} \rceil 65 \leq d_5$ implies that $T \geq 540$. In the second case, we have that $T \geq 96$ and $R_5 = 55 + \lceil \frac{R_5}{2T} \rceil 70 \leq d_5$ implies that $T \geq 555$. So, the best case is to allocate p_5 to partition V_1 (so, $T = 540$ and $U = 0.78$).

This best-fit allocation strategy aims at maximizing the processor utilization. As shown in the overflow and underflow analyses, the task allocation strategy affects the buffer sizes computation. Hence, an allocation strategy that reduces the buffering requirements can be more suitable for systems with strong memory constraints. For instance, a task p_k could be assigned to a partition so that the total sum of sizes of channels that connect p_k to the already assigned tasks is minimized.

For global FP scheduling, we use the schedulability test proposed in [26] (Section 1.4.3) assuming that there is a unique priority region. Again, we will consider only connected dataflow graphs. The algorithm consists in arranging tasks by the decreasing order of priorities. Then, for each task p_k and starting from the solution found in the previous step, we compute the best solution that satisfies $R_k \leq d_k$. So, we do not have to check the schedulability of the already processed tasks (thanks to Lemma 3.1).

Lemma 3.1. If $\mathcal{T}' \geq \mathcal{T}$, then the value of R_k at $\mathcal{T}'$ is smaller than its value at $\mathcal{T}$.

Proof: Since $R_k = C_k + \left\lfloor \frac{1}{M} \sum_{w_i < w_k} I_i(R_k) \right\rfloor$, it is sufficient to prove that $\forall L : I_i(L)$ is smaller at $\mathcal{T}'$ than at $\mathcal{T}$. Recall that $I_i(L)$ is an upper bound on the interference of task p_i on task p_k in an interval of length L. This interference is higher for small periods and deadlines of p_i. Hence, $I_i(L)$ at point $\mathcal{T}'$ is smaller than $I_i(L)$ at point $\mathcal{T}$. □

Example 3.6. We again consider the task set in Example 3.3 with priority assignment p_1 p_2 p_5 p_4 p_3 and $M = 2$. For actor p_1, we have $R_1 = 65$. Hence, $T = 96$. For actor p_2, we have that $T \geq 96$ and $T \geq R_2 = 70 + \left\lfloor \frac{1}{2} \min\{R_2 - 69, W_1\} \right\rfloor$ such that $W_1 = N_1 65 + \min\{65, R_2 - T - 95 - N_1 T\}$ and $N_1 = \left\lfloor \frac{R_2 + T - 95}{T} \right\rfloor$. Hence, $T = 96$, etc.

3.3 EDF scheduling

In this section, we present the symbolic EDF schedulability analysis of dataflow graphs for uniprocessor and multiprocessor systems. We also describe the lock-free implementation of the totally ordered communication strategy.

3.3.1 Deadlines adjustment

In FP scheduling, adjusting priorities of actors can be used to reduce the buffering requirements. Similarly, adjusting priorities of jobs in EDF scheduling could improve the buffering requirements, as shown in the previous chapter. Let $d_i[j]$, $R_i[j] = r_i + (j-1)\pi_i$, and $D_i[j] = R_i[j] + d_i[j]$ be the relative deadline, the release time, and the absolute deadline of job $p_i[j]$, respectively. One way to ensure that job $p_i[j]$ has a higher priority

than job $p_k[j']$ (i.e. $\omega_i[j] < \omega_k[j']$) without using lock-based synchronizations is to make sure that $D_i[j] < D_k[j']$ if $R_i[j] = R_k[j']$ and $\mathsf{ID}(p_i) > \mathsf{ID}(p_k)$; and $D_i[j] \leq D_k[j']$ otherwise.

Totally ordered communication

The totally ordered communication strategy was used in abstract EDF schedules contruction to eliminate preemptions and hence obtain more accurate buffer sizes. The precedences imposed by this strategy can be encoded as deadlines adjustments. Figure 3.8 represent a $(5, 3, 4)$-affine relation between producer p_i and consumer p_k of a channel e. Arrows represent the precedence relation. For instance, the scheduling constraint "$p_k[4]$ precedes $p_i[4]$" can be encoded as $D_k[4] < D_i[4]$ (assuming that $\mathsf{ID}(p_i) < \mathsf{ID}(p_k)$) or as $D_k[4] = \min\{D_k[4], D_i[4] - 1\}$.

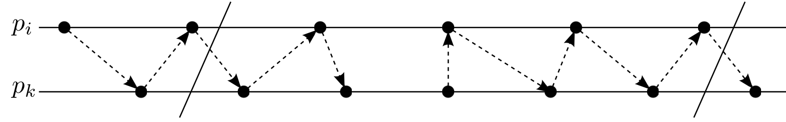


Figure 3.8: Totally ordered communication strategy.

Let S_i be the set of neighbors of p_i in the dataflow graph; i.e. $\forall p_k \in S_i$: p_k communicates with p_i and therefore p_k and p_i are affine-related. Let $p_k[\mathsf{safr}_{i,k}(j)]$ be the earliest job of p_k that must be preceded by job $p_i[j]$; i.e. $p_k[\mathsf{safr}_{i,k}(j)]$ can only start when $p_i[j]$ completes its execution. So, $\mathsf{cbef}_{k,i}(\mathsf{safr}_{i,k}(j)) \geq j$. Figure 3.8 gives an intuition on how to compute $\mathsf{safr}_{i,k}(j)$. There are mainly five cases, illustrated in Figure 3.9 where $j' = \mathsf{safr}_{i,k}(j)$. The totally ordered communication is just an example of communication strategies; one can imagine more possible strategies.

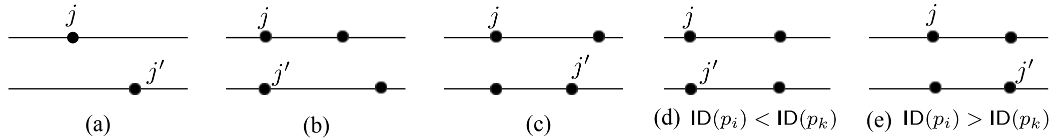


Figure 3.9: $\mathsf{safr}_{i,k}$ function.

Let d_i^* be the user-provided deadline of actor p_i (it could be the implicit deadline) and $\zeta_{i,k}(j) = 1$ if $D_i[j]$ must be *strictly* less than $D_k[\mathsf{safr}_{i,k}(j)]$, and $\zeta_{i,k}(j) = 0$ otherwise. The new deadline of job $p_i[j]$ can be computed as follows.

$$D_i[j] = \min_{p_k \in S_i} \{R_i[j] + d_i^*, D_k[\mathsf{safr}_{i,k}(j)] - \zeta_{i,k}(j)\} \tag{3.9}$$

If we consider relative deadlines, then Equation 3.9 can be written as

$$\begin{aligned} d_i[j] &= D_i[j] - R_i[j] \\ &= \min_{p_k \in S_i} \{d_i^*, d_k[\mathsf{safr}_{i,k}(j)] - \zeta_{i,k}(j) + R_k[\mathsf{safr}_{i,k}(j)] - R_i[j]\} \end{aligned}$$

It is worth mentioning that even if p_i belongs to a cycle in the dataflow graph, $D_i[j]$ does not depend on itself. Therefore, it is possible to compute adjusted deadlines using simple propagation of new values. Furthermore, since each affine relation is an ultimately periodic activation relation, the deadlines of jobs of each actor are ultimately periodic.

Example 3.7. Consider the UCSDF graph in Figure 2.14 (p. 79) with affine relations $p_1 \xrightarrow{(1,1,2)} p_2 \xrightarrow{(3,-1,1)} p_3 \xrightarrow{(2,-1,3)} p_1$. As shown in Figure 3.10, the hyperperiod of this schedule consists of two activations of p_1, one activation of p_2, and three activations of p_3. Deadlines computation will be limited to one hyperperiod. We have that $\pi = [T, 2T, \frac{2}{3}T]$ and we suppose that the user-provided deadlines are $d^* = [\pi_1 - 10, \frac{\pi_2}{2}, \pi_3]$ and that $\mathsf{ID}(p_1) < \mathsf{ID}(p_2) < \mathsf{ID}(p_3)$. The new deadlines are computed as follows.

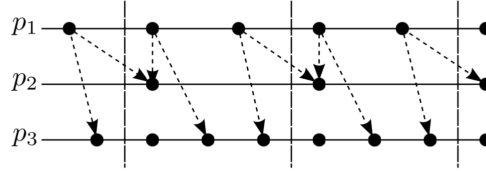


Figure 3.10: Deadlines computation example.

$$\begin{aligned}
&d_1[1] = \min\{d_1^*, d_3[1] + R_3[1] - R_1[1]\} = T - 10\\
&d_1[2] = \min\{d_1^*, d_2[1], d_3[3] + R_3[3] - R_1[2]\} = T - 10\\
&d_1[3] = \min\{d_1^*, d_3[4] + R_3[4] - R_1[3]\} = T - 10\\
&d_2[1] = \min\{d_2^*, d_1[3] + R_1[3] - R_2[1], d_3[3] + R_3[3] - R_2[1]\} = \min\{T, 2T - 10\}\\
&d_3[1] = d_3^* = \tfrac{2}{3}T\\
&d_3[2] = \min\{d_3^*, d_1[2] - 1, d_2[1] - 1\} = \min\{\tfrac{2}{3}T, T - 11\}\\
&d_3[3] = \min\{d_3^*, d_1[3] + R_1[3] - R_3[3]\} = \min\{\tfrac{2}{3}T, \tfrac{4}{3}T - 10\}\\
&d_3[4] = d_3^* = \tfrac{2}{3}T
\end{aligned}$$

The possible deadline regions are depicted in Table 3.2. So, if $C = [65, 70, 95]$, then $T \geq \frac{242.5}{M}$. Hence, if $M = 1$, then $\forall p_i : d_i = d_i^*$.

Table 3.2: Deadline regions of Example 3.7.

$T \in$	$]10, 15]$	$]15, 33]$	$]33, \infty[$
d_1	$(T - 10)^\omega$	$(T - 10)^\omega$	$(T - 10)^\omega$
d_2	$(T)^\omega$	$(T)^\omega$	$(T)^\omega$
d_3	$\frac{2}{3}T(T - 11\ \frac{4}{3}T - 10\ \frac{2}{3}T)^\omega$	$\frac{2}{3}T(T - 11\ \frac{2}{3}T\ \frac{2}{3}T)^\omega$	$(\frac{2}{3}T)^\omega$

Approximate deadlines

Most of EDF schedulability tests assume that a task has a unique deadline. To fit in this model, we may consider the following two simple solutions.

1. For each actor p_i, we compute a minimum deadline $d_i = \min_j d_i[j]$. These deadlines are used *just* in the symbolic schedulability analysis since if the system is schedulable with the minimum deadlines, then it is schedulable with the original deadlines. This is true for sustainable schedulability tests (see [50]) such as the processor demand analysis used in the sequel. However, the original deadlines must be used in the implementation since the minimum deadlines do not guarantee the totally ordered communication strategy. In the previous example, we may take $d_3 = T - 11$ when $T \in]15, 33]$.

2. A unique deadline is computed for each task p_i as $d_i = \min_j d_i[j]$. Hence,

$$\begin{aligned} d_i &= \min_j d_i[j] = \min_j \min_{p_k \in S_i} \{d_i^*, d_k[\mathsf{safr}_{i,k}(j)] - \zeta_{i,k}(j) + R_k[\mathsf{safr}_{i,k}(j)] - R_i[j]\} \\ &= \min_{p_k \in \mathcal{S}_i} \{d_i^*, \min_j \{d_k[\mathsf{safr}_{i,k}(j)] - \zeta_{i,k}(j) + R_k[\mathsf{safr}_{i,k}(j)] - R_i[j]\}\} \\ &\geq \min_{p_k \in \mathcal{S}_i} \{d_i^*, d_k + \min_j \{-\zeta_{i,k}(j) + R_k[\mathsf{safr}_{i,k}(j)] - R_i[j]\}\} \\ &\quad \text{(because } \min_j d_k[\mathsf{safr}_{i,k}(j)] \geq \min_j d_k[j] = d_k\text{)} \end{aligned}$$

So, we only need to compute a lower bound on $\min_j \{-\zeta_{i,k}(j) + R_k[\mathsf{safr}_{i,k}(j)] - R_i[j]\}$. That depends on the affine relation between p_i and p_k. If the affine relation contains synchronous activations (i.e. $\gcd\{n, d\} = 1$ assuming that the affine relation is under canonical form) and $\mathsf{ID}(p_i) > \mathsf{ID}(p_k)$, then $\min_j \{-\zeta_{i,k}(j)\} = -1$; otherwise $\min_j \{-\zeta_{i,k}(j)\} = 0$. Similarly, from the affine relation, it is possible to compute $\min_j \{R_k[\mathsf{safr}_{i,k}(j)] - R_i[j]\}$. Let us put $\Delta_{i,k} = \min_j \{-\zeta_{i,k}(j) + R_k[\mathsf{safr}_{i,k}(j)] - R_i[j]\}$. Therefore, we have that

$$d_i = \min_{p_k \in S_i} \{d_i^*, d_k + \Delta_{i,k}\} \tag{3.10}$$

If we put $X = [d_1, d_2, \dots, d_N]$, then approximate deadlines computation can be written as $X = F(X)$. So, we need to compute the greatest fixed point of the function F. Since F is a monotone function, the fixed point, *if any*, can be found by computing the sequence $X^0, X^1 = F(X^0), X^2 = F(X^1), \dots$ until stabilization such that X^0 is the vector where $\forall p_i \in P : d_i = d_i^*$.

Example 3.8. Let us consider the graph in Example 3.7. Using Equation 3.10, we obtain $d_1 = \min\{d_1, d_2 + T, d_3 + \frac{T}{3}\}$, $d_2 = \min\{d_2, d_1 + T, d_3 + \frac{2}{3}T\}$, and $d_3 = \min\{d_3, d_1 - 1, d_2 - 1\}$. The solutions of this equation system are presented in Table 3.3.

Property 3.2. Deadlines are monotone over the $\mathcal{T}$-space.

This property means that for two points $\mathcal{T}$ and $\mathcal{T}'$ in the $\mathcal{T}$-space such that $\mathcal{T} \leq \mathcal{T}'$, the deadline of any job at point $\mathcal{T}'$ is greater than or equal to its deadline at point $\mathcal{T}$.

Table 3.3: Fixed-point computation of deadlines.

$T \in$	$]10, 16]$	$]16, 33]$	$]33, \infty[$
d_1	$T - 10$	$T - 10$	$T - 10$
d_2	$\frac{5}{3}T - 11$	T	T
d_3	$T - 11$	$T - 11$	$\frac{2}{3}T$

Impact on throughput

Let $\mathcal{T}^*$ be the point that gives the best processor utilization in the initial $\mathcal{T}$-space (i.e. using the necessary schedulability test $U \leq M$). Any increase on the lower bound of T_j, due to deadlines adjustment, potentially decreases the processor utilization if $T_j^l > \mathcal{T}_j^*$. The lower bound could be increased after adjusting the deadlines in order to meet the constraint $C_i \leq d_i[j]$. One simple solution is to not consider in the deadlines adjustment any precedence that may increase the lower bounds of components T_j.

3.3.2 Uniprocessor scheduling

We firstly consider the case where the graph of affine relations is connected. Thus, $\forall p_i \in P : \pi_i = \alpha_i T$. If the deadlines are equal to periods, we use the schedulability test $U \leq 1$ since it is an exact EDF schedulability test. The solution is therefore equal to $T = \left\lceil \frac{T^l}{B} \right\rceil B$ (computation of T^l has already considered the constraint $U \leq 1$). If the deadlines are constrained, then we use the QPA schedulability test (Section 1.4.2). The basic symbolic schedulability test consists in using QPA at each point $T_k = \left\lceil \frac{T^l}{B} \right\rceil B + kB$ in the interval $[T^l, T^u]$ in the increasing order till reaching the first feasible solution. This time consuming approach can be improved knowing that "***when we increase T, deadlines and period are stretched while execution times remain constants***".

Our symbolic schedulability analysis of dataflow graphs consists in incorporating the search of the minimum T that ensures EDF schedulability into the QPA algorithm. Algorithm 5 represents the symbolic QPA algorithm. Let $L(T), U(T)$, and $h^T(t)$ denote respectively the values L, U, and $h(t)$ for a given T. Recall that L is the feasibility bound and h is the processor demand function.

Starting from the minimum value of T (i.e. T_0), SQPA performs the QPA analysis in the interval $[0, L(T_0)]$. This first iteration leads either to $h^{T_0}(t) \leq \min\{d\}$ or to a deadline miss, i.e. $h^{T_0}(t) > t$ (Figure 3.11). In the first case, the task system is schedulable and the algorithm returns T_0.

In the second case, assume that the deadline miss occurs at d^* (i.e. $h^{T_0}(d^*) > d^*$). In this case, T must be increased to T_1 (instruction `k++;`). According to Lemma 3.2, $L(T_1) \leq L(T_0)$. Hence, the verification can restart from $L(T_1)$ instead of $L(T_0)$. But, according to Lemma 1.1 (p. 41), we have that $\forall t \in [h^{T_0}(d^*), L(T_0)] : h^{T_0}(t) \leq t$. Since Lemma 3.3 implies that $\forall t \in [h^{T_0}(d^*), L(T_0)] : h^{T_1}(t) \leq h^{T_0}(t) \leq t$, the verification process for T_1 can restart from $\min\{L(T_1), h^{T_0}(d^*)\}$. This process is repeated until the termination condition is reached or T exceeds the upper bound T^u.

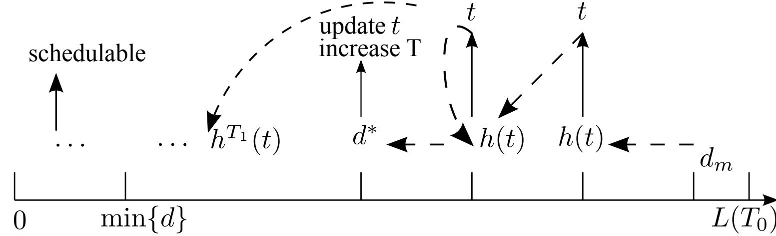


Figure 3.11: Illustration of SQPA.

Algorithm 5: SQPA algorithm

$k = 0;\ t = \max\{d | d \leq L(T_k)\}$;
while $h(t) > \min\{d\}$ **do**
  if $h(t) < t$ **then** $t = h(t)$;
  else if $t == h(t)$ **then** $t = \max\{d | d < t\}$;
  else
    k++;
    if $T_k > T^u$ **then return** task set not schedulable;
    $t = \min\{h^{T_{k-1}}(t), \max\{d | d \leq L(T_k)\}\}$;
return T_k;

Lemma 3.2. *If* $T \leq T'$*, then* $L(T) \geq L(T')$*.*

Proof: We have that L is equal to synchronous busy period which can computed with $L^{m+1} = \sum_{p_i \in P} \left\lceil \frac{L^m}{\pi_i} \right\rceil C_i$ where $L^0 = \sum_{p_i \in P} C_i$. Clearly, $\forall m : L^m(T') \leq L^m(T)$. Hence, $L(T') \leq L(T)$. □

Lemma 3.3. *If* $T \leq T'$*, then* $\forall t : h^T(t) \geq h^{T'}(t)$*.*

Proof: We have that $h^T(t) = \sum_{p_i \in P} h_i^T(t)$ such that $h_i^T(t) = \sum_{D_i[j] \leq t} C_i$. According to Property 3.2, a given absolute deadline occurs earlier for T than for T'. Therefore, $\forall l, h_i^T(t) \geq h_i^{T'}(t)$. □

SQPA algorithm has a pseudo-polynomial complexity since it checks in the worst-case scenario all the deadlines in interval $[0, L(T_0)]$ as a standard processor-demand analysis may do.

Example 3.9. We take the same example as in Example 3.7 with $C = [65, 70, 95]$. We have that $U = \frac{242.5}{T}$, $T \geq 243$, and $B = 3$.

- $T = 243, L = 485$

$t = 476, h(t) = 390$ $t = 390, h(t) = 325$
$t = 325, h(t) = 325$ $t = 324, h(t) = 325$ (deadline miss; increase T)

- $T = 246, L = 485$

$t = 325, h(t) = 230$ $t = 230, h(t) = 95$ (return $T = 246$)

So, the maximum processor utilization that can be achieved with the user-provided deadlines is equal to $U = 0.985$.

The second case is when the graph of affine relations is disconnected. We propose a DF-B&B SQPA search algorithm (Algorithm 6). Procedure $\texttt{SQPA}^*(\mathcal{T}, t)$ performs QPA (i.e. testing $h(t) \leq t$ in backward manner) for a given value $\mathcal{T}$ starting from point $\min\{t, L(\mathcal{T})\}$. It ends when there is a miss (i.e. this procedure does not increase periods) or if the system is schedulable at point $\mathcal{T}$. When $U(\mathcal{T}) > 1$, the procedure indicates the miss immediately. To better explain the search algorithm, we consider the following example.

Algorithm 6: DF-B&B SQPA.

```
Procedure main() begin
    T^cur = [T_1^l, ..., T_L^l]; U^cur = 0;
    t = SQPA*(T^cur, L(T^cur));
    if t > min{d} then VisitTree(T^cur, t, 1);
    return T^cur;

Procedure visitTree(T, t, j) begin
    for i = j, j + 1, ..., L, ..., j - 1 do
        T^node <- increase T_i in T;
        if T_i > T_i^u or U(T^node) <= U^cur then prune this node;
        else
            t = SQPA*(T^node, t) ;
            if t <= min{d} then T^cur = T^node; U^cur = U(T^node);
            else VisitTree(T^node, h^{T^node}(t), j mod L + 1);
```

Example 3.10. Let us take the dataflow graph in Example 3.2 with $C = [20, 30, 10, 15, 10]$ and $d = [\pi_1 - 30, \frac{\pi_2}{2} - 5, \pi_3, \pi_4 - 5, \frac{4}{5}\pi_5 - 10]$. We have that $\pi = [T_1, 2T_1, \frac{2}{3}T_1, T_2, \frac{T_2}{4}]$. We suppose that the initial $\mathcal{T}$-space is defined by $T_1 \geq 84, T_2 \geq 100, B_1 = 12$, $B_2 = 20$, and $U = \frac{50}{T_1} + \frac{55}{T_2} \leq 1$.

Figure 6 represents the results of the search algorithm where nodes are numbered in the order of appearance. Initially, we have that $\mathcal{T}^{\text{cur}} = [102, 96]$ and $U = 0$. Since $U(\mathcal{T}^{\text{cur}}) > 1$, procedure $\texttt{SQPA}^*$ indicates a miss immediately. From each node, we increase either T_1 or T_2. The first node with $U(\mathcal{T}^{\text{node}}) \leq 1$ is $\mathcal{T}^{\text{node}} = [96, 120]$. At this node, Procedure $\texttt{SQPA}^*$ returns a miss. As in SQPA, If d^* is the missed deadline, then the verification will continue from $\min\{L, h(d^*)\}$ when periods are increased ($h(d^*)$ is denoted by t in Figure 3.12). The first encountered leaf is $\mathcal{T} = [120, 160]$ (i.e. the task set is EDF-schedulable at this point), and U^{cur} and $\mathcal{T}^{\text{cur}}$ must be updated. Node numbered 12 is pruned because $U([144, 120]) < U^{\text{cur}} = 0.806$.

However, this algorithm requires upper bounds T_j^u in order to have a finite search space. This is mainly to ensure that a point such $\mathcal{T} = [\infty, T_2^l]$ or $\mathcal{T} = [T_1^l, \infty]$ cannot

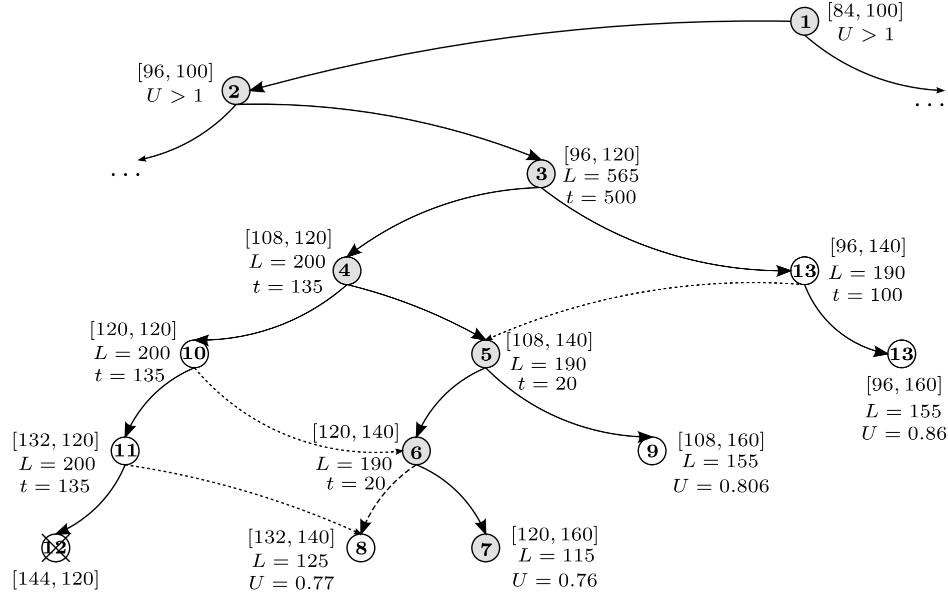


Figure 3.12: Illustration of DF-B&B SQPA.

be the optimum solution. As in DF−B&B SRTA, it is possible to limit the search space to the sub-space dominated by a first admissible solution.

3.3.3 Multiprocessor scheduling

Let us first address the symbolic partitioned EDF scheduling problem. As in FP scheduling, we consider a best fit allocation strategy. Actors are ordered according to some criterion (e.g. non-decreasing order of deadlines in case there is a unique DM priority region). Then, actor p_k is assigned to the best partition. Let $\mathcal{T}_i$ be the value of $\mathcal{T}$ returned by uniprocessor schedulability analysis and which is the best vector for which all partitions $(V_j)_{j \neq i}$ and $V_i \cup \{p_k\}$ are EDF-schedulable each one on a single processor. We assign task p_k to the partition that gives the highest processor utilization $U(\mathcal{T}_i)$. In case of equality, we break the tie by favoring the partition with the minimum utilization $U^i = \sum_{p_j \in V_i \cup \{p_k\}} \frac{C_j}{\pi_j}$ and hence seek for balanced partitions. This best fit allocation strategy aims at maximizing the processor utilization. As shown in the previous chapter, the processor allocation influences the buffer sizes computation. Furthermore, adjusting deadlines to enforce precedences between jobs of two actors does not work if the two actors are allocated to different processors. The approximate gains that come from assigning two actors to the same processor and from totally ordered communication strategy can be deduced using overflow and underflow equations.

An allocation strategy, for connected graphs, that firstly aims at minimizing the buffering requirements (by adjusting implicit deadlines) and secondly balances the processor utilization of partitions consists of the following steps.

1. Parameters n and d of every affine relation are computed using the boundedness

criterion. So, we have that $\forall p_i \in P : \pi_i = \alpha_i T$.

2. Let $G = (V, E)$ be an undirected graph where nodes represent actors and edges represent affine relations. Each node v_i is associated with a weight that represents the utilization of actor p_i; i.e. $w(v_i) = \frac{C_i}{\alpha_i}$. The weight of edge $e_{i,k} = (v_i, v_k)$, denoted by $w(e_{i,k})$, is equal to zero if enforcing precedences between p_i and p_k may jeopardize the processor utilization factor, and equal to the approximate gain that comes from adjusting the deadlines otherwise.

3. The graph G should be partitioned into M balanced partitions $(V_i)_{i=1,M}$; that is, $\forall i, j : |w(V_i) - w(V_j)|$ is minimal such that $w(V_i) = \sum_{v \in V_i} w(v)$. Furthermore, the partitioning must minimize the total weight of edges connecting different partitions. The reason behind this requirement is that adjusting deadlines will not ensure precedences between actors allocated to different processors. This M-partitioning problem is well known in graph theory; for instance, we use the SCOTCH tool [59] to solve the problem.

4. Once each actor is assigned to a processor, overflow and underflow analyses can be performed to compute parameter φ of each affine relation.

5. Computation of symbolic deadlines of tasks in each partition.

6. The SQPA algorithm is then applied on each partition V_i. The algorithm will return the minimum value T_i that ensures EDF schedulability of the set V_i on a single processor. We need just to take $T = \max_{i=1,M} T_i$.

For global EDF scheduling of connected graphs, we will use the QPA-FFDBF schedulability test. But first, we propose an improvement of the schedulability test using the following lemma.

Lemma 3.4. *If* $\gamma_1 < \gamma_2$ *and* $\mathit{ffdbf}(t, \gamma_1) \geq (\frac{M-(M-1)\gamma_1}{(M-1)\gamma_2}) \sum_{p_i \in P} C_i$*, then* $h(t, \gamma_1) \leq h(t, \gamma_2)$.

Proof: Figure 3.13 depicts the forced-forward demand bound function for two values γ_1 and γ_2 such that $\gamma_1 < \gamma_2$. As shown in that figure, we have clearly that $\forall t : 0 \leq \Delta_i(t) = \mathit{ffdbf}_i(t, \gamma_1) - \mathit{ffdbf}_i(t, \gamma_2) \leq \Delta_i$. Using basic geometry, we have that $\Delta_i = C_i(1 - \frac{\gamma_1}{\gamma_2})$. So,

$$h_i(t, \gamma_2) - h_i(t, \gamma_1) = \frac{\mathit{ffdbf}_i(t, \gamma_1) - \Delta_i(t)}{M - (M-1)\gamma_2} - \frac{\mathit{ffdbf}_i(t, \gamma_1)}{M - (M-1)\gamma_1}$$

Hence,

$$h(t, \gamma_2) - h(t, \gamma_1) = \frac{x}{(M - (M-1)\gamma_2)(M - (M-1)\gamma_1)}$$

s.t. $x = (M-1)(\gamma_2 - \gamma_1)\,\mathit{ffdbf}(t, \gamma_1) - (M - (M-1)\gamma_1) \sum_{p_i \in P} \Delta_i(t)$. Therefore, if $x \geq 0$; i.e.

$$\mathit{ffdbf}(t, \gamma_1) \geq \frac{M - (M-1)\gamma_1}{(M-1)(\gamma_2 - \gamma_1)} \sum_{p_i \in P} \Delta_i(t)$$

then $h(t, \gamma_1) \leq h(t, \gamma_2)$. But, $\forall t : \sum_{p_i \in P} \Delta_i(t) \leq \sum_{p_i \in P} \Delta_i$. □

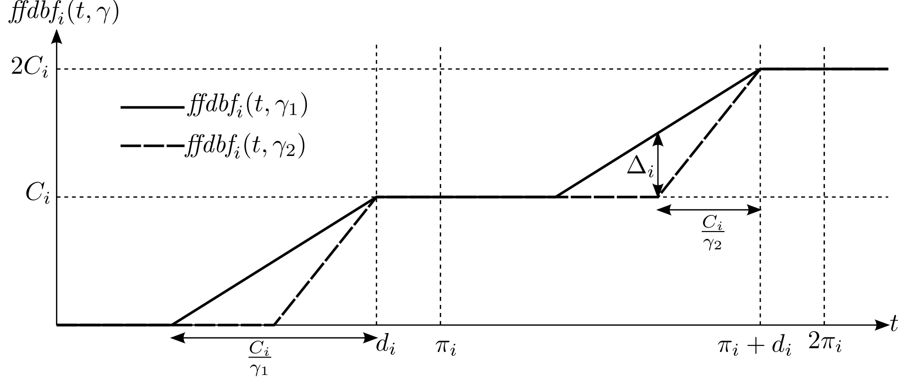


Figure 3.13: Illustration of *ffdbf*

Lemma 3.4 is used as follows. For a given γ_1, if $h(t, \gamma_1) > t$ and $\mathit{ffdbf}(t, \gamma_1) \geq F^* = (\frac{M-(M-1)\gamma_1}{(M-1)\gamma_1}) \sum_{p_i \in P} C_i$, then it does not matter if we increase γ_1 to γ_2 since $h(t, \gamma_2)$ will be also greater than t. Note that we used γ_1 in the dominator of F^* instead of γ_2.

Algorithm 7 represents the symbolic QPA-FFDBF algorithm for global EDF scheduling of connected graphs. Let $\gamma_{\min}(T)$, and $\gamma_{\max}(T)$ denote respectively $\gamma_{\min} = \mu^*$, and $\gamma_{\max} = \frac{M-U}{M-1}$ for a given T. Let $F^*(\gamma)$ denote the value of F^* for a given value of γ. If $h(t, \gamma) > t$, then we have to increase either γ or T according to Lemma 3.4 and Theorem 1.2 (p. 42). As for SQPA, we note the following results.

Lemma 3.5. If $T \leq T'$, then $\forall t : h^T(t, \gamma) \geq h^{T'}(t, \gamma)$.

Proof: We have that $h^T(t, \gamma) = \frac{\sum_{p_i \in P} \mathit{ffdbf}_i^{\,T}(t,\gamma)}{M-(M-1)\gamma}$. It is quite easy to prove that $\mathit{ffdbf}_i^{\,T'}(t, \gamma) \leq \mathit{ffdbf}_i^{\,T}(t, \gamma)$. Indeed, periods and deadlines are larger at point T' than at point T. You can easily notice in Figure 3.13 that the value of the forced-forward demand bound function decreases when periods and deadlines are increased. □

If d^* is the first deadline for which $h^T(d^*, \gamma) > d^*$, then we have that $\forall t \in [h^T(d^*, \gamma), L(T)] : h^{T'}(t, \gamma) \leq h^T(l, \gamma) \leq t$. For given values T_{k+1} and γ, we take $\text{Prev}(\gamma) = h^{T_k}(d^*, \gamma)$.

Lemma 3.6. If $T \leq T'$, then $L(T) \geq L(T')$.

Proof: If there is a unique deadline region and $\forall p_i : \frac{d_i}{\pi_i}$ is monotone over the $\mathcal{T}$-space, then $L = \frac{\sum_{p_i \in P} (\pi_i - d_i)U_i}{M-(M-1)\gamma-U}$ satisfies the property; otherwise $L = \frac{\sum_{p_i \in P} C_i}{M-(M-1)\gamma-U}$ (from [11]) satisfies the property. □

Thanks to these observations, for a given value γ, it is not necessary to recheck deadlines in the interval $[L(T'), \text{Prev}(\gamma)]$ when T is increased to T'.

It is worth mentioning that computation of $\mathcal{T}^l$ must consider the constraint $\mu^* < \frac{M-U}{M-1}$. If the graph of affine relation is disconnected, then a branch and bound technique like the one proposed for SQPA can be used with SQPA-FFDBF.

Algorithm 7: SQPA-FFDBF algorithm

$k = 0;\ \gamma = \gamma_{\min}(T_k);\ t = L(T_k);$
while $h(t,\gamma) > \min\{d\}$ **do**
  if $h(t,\gamma) \leq t$ **then** $t = \min\{h(t,\gamma), \max\{d|d < t\}\};$
  else
    if $\mathit{ffdbf}(t,\gamma) \geq F^*(\gamma) \vee \gamma + \epsilon \geq \gamma_{\max}(T_k)$ **then**
      k++;
      if $T_k > T^u$ **then return** task set unschedulable;
      $\gamma = \gamma_{\min}(T_k);$
    else $\gamma = \gamma + \epsilon;$
    $t = \min\{L(T_k), \mathrm{Prev}(\gamma)\};$
return T_k;

Example 3.11. Let us consider the task set in Example 3.3: $\pi = [T, 2T, \frac{2}{3}T, \frac{4}{3}T, \frac{T}{3}]$, $C = [65, 70, 95, 60, 55]$, $d = [\pi_1 - 30, \frac{\pi_2}{2}, \frac{3}{4}\pi_3 - 10, \frac{\pi_4}{2}, \pi_5 - 60]$, and $U = \frac{452.5}{T}$. If $M = 2$, then the initial $\mathcal{T}$-space is defined by $T \geq 348$ and $B = 6$. We have that $\gamma_{\min}(T) = \max\{\frac{190}{T-20}, \frac{165}{T-180}\}$ and $\gamma_{\max}(T) = 2 - \frac{452.5}{T}$. Constraint $\mu^* < \frac{M-U}{M-1}$ implies that $T \geq 384$. We take $\epsilon = 0.002$ and the initial value of γ is rounded up (e.g. $0.6893 \rightarrow 0.690$) in order to pass by the same values of γ in the iterations.

□ $T = 384,\ \gamma_{\min} = 0.8088,\ \gamma_{\max} = 0.8216$
• $\gamma = 0.810,\ \ L = 10504$
$t = 10504, h(t,\gamma) = 10425.21 \qquad t = 10425, h(t,\gamma) = 10363.86 \quad \ldots$
$t = 6584, h(t,\gamma) = 6584.03$: *miss*.
Increasing γ will not solve the problem since $\mathit{ffdbf}(t,\gamma) = 7835.0 > F^* = 506.85$. Hence, T must be increased.
□ $T = 390,\ \gamma_{\min} = 0.7857,\ \gamma_{\max} = 0.8397$
• $\gamma = 0.786,\ \ L = 2270$
$t = 2265, h(t,\gamma) = 2173.49 \quad \ldots \quad t = 751, h(t,\gamma) = 751.23$: *miss* and $\mathit{ffdbf}(t,\gamma) > F^*$.
⋮
□ $T = 408,\ \gamma_{\min} = 0.7236,\ \gamma_{\max} = 0.8909$
• $\gamma = 0.724,\ \ L = 718$: *schedulable* and $\frac{U}{M} = 0.554$

3.4 Conclusion

In this chapter, we have presented the necessary symbolic schedulability analyses of abstract schedules that aim at either minimizing the buffering requirements or maximizing the processor utilization. We have presented the symbolic schedulability analysis for uniprocessor and multiprocessor systems with respect to two real-time scheduling policies: fixed-priority scheduling and earliest-deadline first scheduling.

Chapter 4

Experimental validation

Contents

4.1 Performance comparison: ADFG vs DARTS 113
4.1.1 Throughput 115
4.1.2 Buffering requirements 118
4.2 Symbolic schedulability analysis 121
4.2.1 EDF scheduling 122
4.2.2 Fixed-priority scheduling 125
4.3 Application: Design of SCJ/L1 systems 127
4.3.1 Concurrency model of SCJ/L1 128
4.3.2 Dataflow design model 129
4.4 Conclusion 131

In this chapter, we present the results obtained by the scheduling algorithms on a set of real-life stream processing applications and randomly generated dataflow graphs w.r.t. buffer minimization and throughput maximization. Furthermore, we compare the complexity of our symbolic schedulability analyses with that of basic enumerative solutions. We refer to the implementation of our scheduling algorithms as the ADFG tool. We will also briefly present a graphical editor for automatic synthesis of SCJ Level 1 applications from UCDF graph specifications.

4.1 Performance comparison: ADFG vs DARTS

In a first place, we compare our scheduling tool (the ADFG tool) with the DARTS tool (an implementation of the scheduling approach presented in [9]). In the best of our knowledge, DARTS is the only existing tool for *real-time* scheduling of (C)SDF graphs. However, it can handle only *acyclic* connected graphs. Therefore, we use the same acyclic benchmarks (see Table 4.1) presented in more details in [9]. They are SDF and CSDF graphs collected from many sources (the StreamIt benchmark [190], the SDF^3 tool [182],

Application	#nodes	#edges	B	σ	$\Theta(G)$	$U^{\max}$	`minBS`
Serpent	120	128	1	129020	2.99×10^{-4}	38.675	14249
Fast Fourier transform (FFT)	17	16	1	141058	8.31×10^{-5}	11.723	8192
MPEG2 video	23	26	1	56737	1.30×10^{-4}	7.387	6274
Digital Radio Mondiale receiver	4	3	768	47207.8	1.24×10^{-5}	2.927	2698
Channel vocoder	55	70	1	1186025	2.81×10^{-5}	33.409	2618
Data Encryption Standard (DES)	53	60	1	18034	9.76×10^{-4}	17.611	2564
Discrete cosine transform (DCT)	8	7	1	121672	2.10×10^{-5}	2.555	1792
Satellite receiver	22	26	220	188.12	9.45×10^{-4}	4.275	1542
H.263 video decoder	4	3	1	1107.24	3.01×10^{-6}	1.98	1189
Filterbank for signal processing	85	99	1	364268	8.84×10^{-5}	32.201	680
Vocoder	114	146	1	25417	1.10×10^{-4}	2.791	668
Multi-channel beamformer	57	70	1	130033	1.97×10^{-4}	25.617	254
Bitonic Parallel Sorting	40	46	1	930	1.05×10^{-2}	9.789	151
Software FM radio with equalizer	43	53	1	18828	6.97×10^{-4}	13.129	57
Heart pacemaker	4	3	320	611	3.12×10^{-3}	1.909	42
MP3 audio decoder	14	18	1	6105381	2.68×10^{-7}	3.271	20
Data modem	6	5	1	2.93	6.25×10^{-2}	2.937	16
CD-to-DAT rate converter	6	5	80	16.36	8.50×10^{-4}	4.227	5

Table 4.1: Real-life stream processing benchmarks.

and some research papers). The number of nodes in a graph is denoted by `#nodes`, the number of channels is denoted by `#edges`, the maximum throughput (i.e. the self-timed throughput) is denoted by $\Theta(G)$, while the minimum buffering requirements that can be obtained by a *static-periodic* schedule is denoted by `minBS`. The maximum throughput and the minimum buffering requirements are computed after disabling auto-concurrency. Minimum buffering requirements are computed using the SDF^3 tool. The throughput of an actor p_i is taken as $\Theta(p_i) = \Theta(G)\vec{r}(i)$. Then, the maximum processor utilization is taken as $U^{\max} = \sum\limits_{p_i \in P} \Theta(p_i)C_i$.

Definition 4.1. Let $e = (p_i, p_k, u_1 v_1^\omega, u_2 v_2^\omega)$ be a channel. Channel e is said to be a matched I/O rates channel if $\frac{\|v_1\|}{|v_1|} \approx \frac{\|v_2\|}{|v_2|}$. It is said to be a perfectly matched I/O rates channel if $\frac{\|v_1\|}{|v_1|} = \frac{\|v_2\|}{|v_2|}$.

If all the channels in a graph are perfectly matched I/O rates channels, then factor B (computed in the symbolic schedulability analysis) will be equal to 1. Most of the applications in Table 4.1 (except for `MP3 decoder`, `cd2dat-csdf`, `Satellite`, `Receiver`, and `Pacemaker`) consist of perfectly matched I/O rates channels. This kind of graphs does not allow us to demonstrate the reduction of buffering requirements that results from priority assignment or deadlines adjustment because of the following observation. According to Table 2.1, the approximate gain that comes from switching the priorities of the producer and the consumer is given as $\frac{\|v_2\|}{|v_2|} \left| \frac{d-n}{n} \right|$. But, this quantity

is very small in case of matched I/O rates channels (and null in case of perfectly matched I/O channels) since $\frac{d}{n} = \frac{\|v_1\|}{|v_1|}\frac{|v_2|}{\|v_2\|}$. Similar observation holds for deadlines adjustment. To better measure the impact of our techniques, we will use a set of randomly generated (cyclic) connected SDF graphs. The graphs are generated by the SDF^3 tool with the following setting. Production and consumption rates follow a normal distribution with a mean equal to 5 and a variance equal to 4. Worst-case execution times follow a normal distribution with a mean equal to 1000 and a variance equal to 300. The graphs are generated with different numbers of nodes (from 6 to 80) such that the average degree of each node is 2.

In most of the experiments, we will assume implicit deadlines unless it is stated otherwise or the deadlines adjustment technique is used. We will show the results obtained by some of the scheduling algorithms presented in the previous chapter. However, we will focus more on uniprocessor scheduling algorithms.

4.1.1 Throughput

We consider here some of the scheduling algorithms that aim at maximizing the processor utilization regardless of the achieved buffering requirements. Let us first address the uniprocessor scheduling case.

Figure 4.1 shows the obtained results by the ADFG and DARTS tools in case of EDF and FP priority scheduling of the real-life benchmarks. Implicit deadlines are used in order to achieve the highest throughput. Furthermore, the priority assignment policy for FP scheduling is taken as the RM priority assignment. Clearly, our scheduling tool gives (by far) the best processor utilization in case of RM scheduling. This is because our RM schedulability analysis is based on the accurate RTA; while the RM schedulability analysis in the DARTS tool is based on the pessimistic utilization-based test.

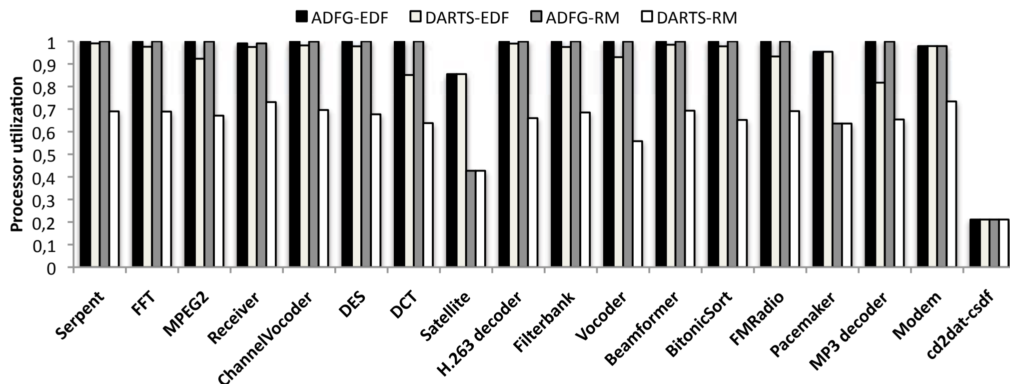


Figure 4.1: Achieved throughput in EDF/RM uniprocessor scheduling.

In case of EDF scheduling, our tool slightly outperforms the DARTS tool. Though both tools use the same exact utilization-based EDF schedulability test ($U \leq 1$), they produce different utilizations due to the implementation details (and not because of the

number of initial tokens in channels). The reason is that the DARTS tool first computes the minimum periods assuming unlimited resources and then scales these periods so that the task set fits on one processor.

Even in EDF scheduling, it is not always possible to achieve an utilization equal to 1. If the system is unschedulable for a given T, then the next value of T depends on B (since $T = \left\lceil \frac{T^l}{B} \right\rceil B + kB$). This explains the low utilization achieved by the `cd2dat-csdf` and `Satellite` applications for instance.

Regarding constrained deadlines, the DARTS tool allows the user to specify deadlines of the form $\forall p_i \in P : d_i = f\pi_i + (1-f)C_i$ such that $f \in]0,1]$. Hence, all the deadlines must be scaled with the same scale factor f. In our tool, the designer has more freedom on which deadlines he would like to constrain as $d_i = a_i\pi_i - b_i$. In Sections 2.3.3 and 2.3.4, we have shown how to compute function cbef in the overflow and underflow analyses. In FP scheduling, user-imposed deadlines has no (direct) impact on the computation of cbef. However, in EDF scheduling, user-provided deadlines may introduce different scheduling precedences than the one deduced from implicit deadlines. Therefore, they must be considered at the abstract schedule construction step.

Let us now address the multiprocessor scheduling case. One important question is "*given a (global or partitioned) scheduling algorithm, what is the minimum number of processors required to achieve the self-timed throughput ?*". A necessary schedulability condition is that $U = \frac{\sigma}{T} \leq M$. But, T must be a multiple of some integer B. Therefore, if $B \geq \sigma$, then even if we increase the number of processors, we cannot reduce the lower bound on T (deduced from $U \leq M$ and $T = kB$). This is why in some cases (e.g. the `satellite` application), it is impossible to achieve the self-timed throughput even with an unlimited number of processors.

In partitioned scheduling, the best processor utilization cannot exceed the one obtained when $M = N$. In this case, the best fit strategy allocates each actor to a different processor. Let $U^{M=N}$ be the obtained utilization and let T^l be the lower bound deduced only from constraint $C_i \leq d_i$. So,

$$U^{M=N} = \frac{\sigma}{\left\lceil \frac{T^l}{B} \right\rceil B}$$

Depending on T^l and B, value $U^{M=N}$ can or cannot be equal to the self-timed utilization.

Application	#nodes	#edges	B	σ	$\Theta(G)$	$U^{\max}$
nodes28	28	63	55125	5895372.96	1.22×10^{-8}	6.938
nodes34	34	64	5040	41300.90	4.90×10^{-10}	4.138
nodes56	56	95	58320	588329.26	3.03×10^{-12}	5.467

Table 4.2: Some randomly generated SDF graphs.

Table 4.2 shows the characteristics of some of the randomly generated SDF graphs. The throughput $\Theta(G)$ is obtained by the SDF^3 tool (auto-concurrency disabled). Figure

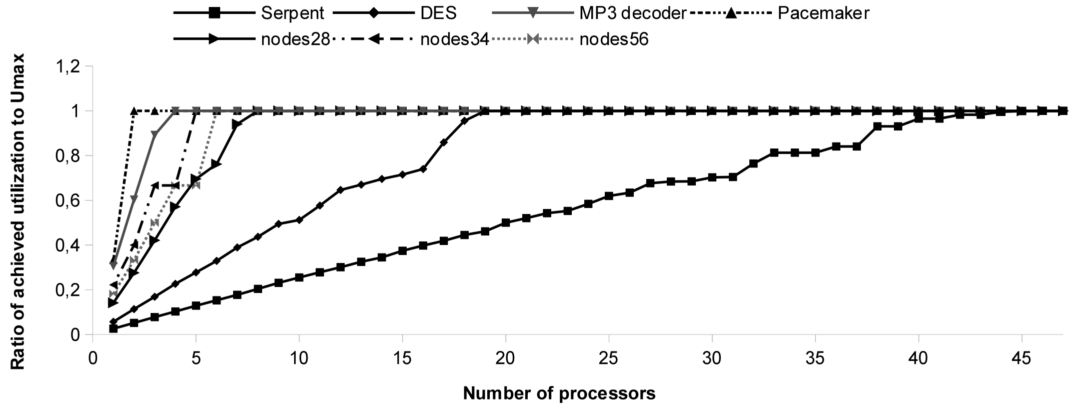


Figure 4.2: Impact of number of processors on the throughput in BF−RM scheduling.

4.2 shows the processor utilization that can be achieved by the best fit partitioned RM algorithm (BF−RM) for different numbers of processors. In BF−RM, actors with high utilization are considered first in case of tie. It is worth mentioning that an optimal multiprocessor scheduling algorithm needs at least $\lceil U^{\max} \rceil$ processors to achieve the self-timed throughput (if it is possible). Clearly, BF−RM is not an optimal algorithm and needs more resources to achieve the maximum throughput. For instance, in case of the Serpent application, BF−RM requires 46 processors to achieve an utilization equal to 38.67.

Regarding partitioned EDF scheduling, we compare our best fit partitioned scheduling algorithm (BF−EDF) with the first fit partitioned scheduling algorithm of the DARTS tool. Figure 4.3 shows the number of processors needed by both algorithms to achieve the self-timed throughput. As one could notice, our algorithm largely outperforms the DARTS tool in all cases. This is due to the symbolic schedulability approach of

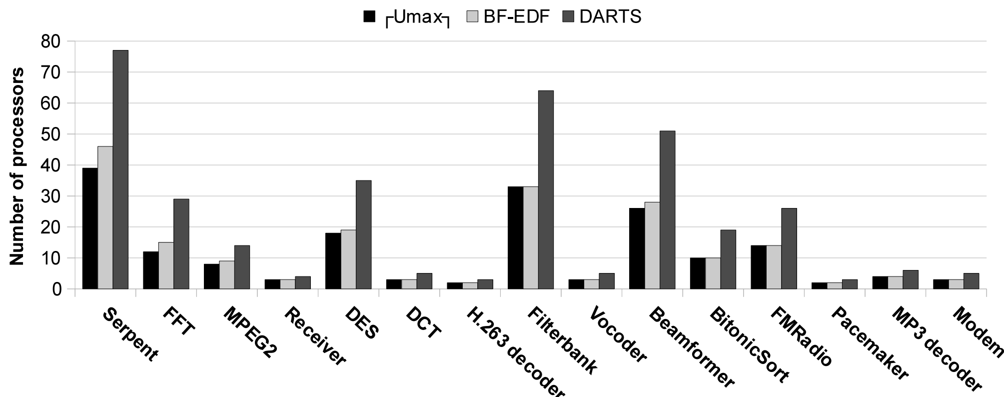


Figure 4.3: Comparison between BF−EDF and the first fit allocation strategy of DARTS.

the DARTS tool and its *first fit* allocation strategy. We note that the ADFG tool has obtained the optimum solution in many cases.

DARTS does not implement any global scheduling algorithm. Therefore, we will compare our global and partitioned scheduling algorithms with each other; for instance, BF−EDF with FFDBF−SQPA. Since FFDBF−SQPA is especially designed for constrained task sets, we assume that $\forall p_i : d_i = \frac{9}{10}\pi_i$. In this case, BF−EDF will apply SQPA on each partition instead of the utilization-based test. Figure 4.4 shows the obtained results for some applications. In this figure, $U^{M=N}$ denotes the maximum utilization which is obtained when dedicating a processor to each actor. Though the forced-forward demand analysis can be considered as one of the best global EDF schedulability analyses [25], this experiment attests the superiority of partitioned EDF schedulability tests over global EDF schedulability tests.

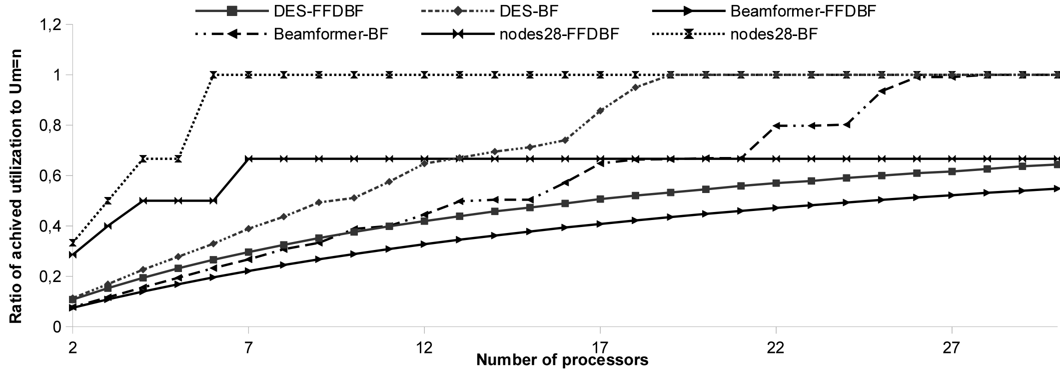


Figure 4.4: Comparison between BF−EDF and FFDBF−SQPA.

4.1.2 Buffering requirements

We consider here the presented scheduling techniques that aim at minimizing the buffering requirements. But, we first compare our tool with the DARTS tool in order to have at least a hint about the accuracy of our linear approximations and machine-independent buffer sizes computation technique. Figure 4.5 represents the ratios of the obtained buffering requirements to the minimum buffering requirements (i.e. minBS). The ADFG tool computes the buffer sizes for uniprocessor EDF scheduling of the benchmarks assuming implicit deadlines. However, the DARTS tool computes buffer sizes under the following hypotheses: (1) An actor consumes its required data at the beginning of a firing and writes all its results at the end. (2) Unlimited number of resources; hence, an actor reads tokens simultaneously with each release. Therefore, the computed buffer sizes are not overflow-safe for uniprocessor scheduling since a consumer can be delayed because of limited resources.

Both tools manage to get the minimum buffering requirements for most of the applications with perfectly matched I/O rates channels (e.g. FFT and Beamformer). However, for applications with mis-matched I/O rates channels (e.g. Satellite,

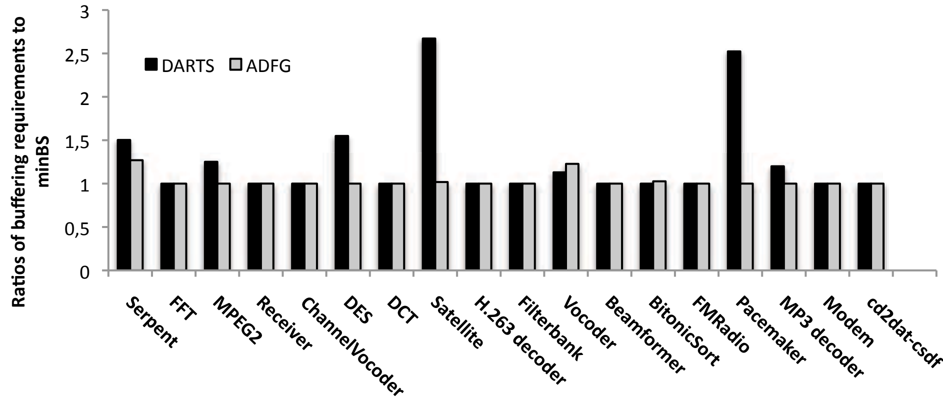


Figure 4.5: Comparison between DARTS and ADFG in terms of buffering requirements.

Pacemaker), our tool largely outperforms the DARTS tool. In average, our tool improves the buffering requirements by 11% compared to the DARTS tool.

Let us now measure the improvement on buffering requirements that can be achieved by the totally ordered communication strategy in uniprocessor EDF scheduling. As we mentioned before, this technique does not improve sizes of perfectly matched I/O rates channels. Thus, we conduct our experiment on the set of randomly generated SDF graphs. Figure 4.6 shows the improvement on the buffering requirements. We found that the totally ordered communication strategy improves the buffering requirements in average by 40% with a very low standard deviation (0.05). However, this improvement comes at the price of a throughput decline with an average equal to 20.95% (Figure 4.6). In 50% of the graphs, the deadlines adjustment improves the buffering requirements without affecting the throughput. These are graphs with highly mis-matched I/O channels. Indeed, when factor B is too large (compared to σ), adjusting deadlines

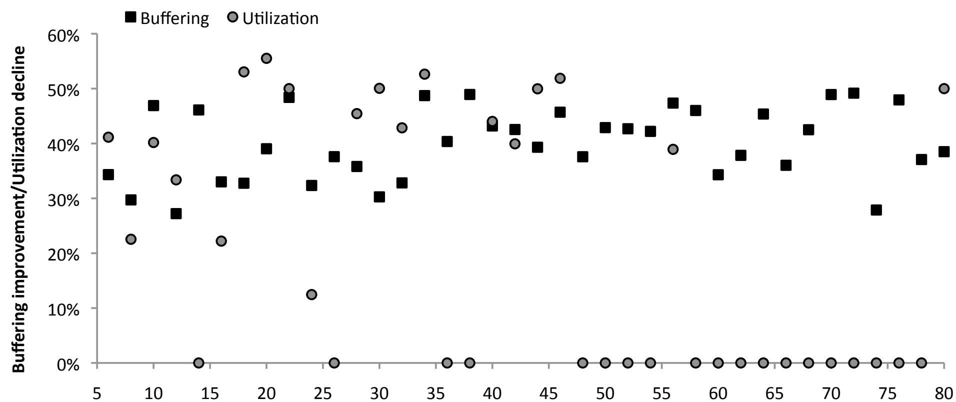


Figure 4.6: Impact of the totally ordered communication strategy on the buffering requirements.

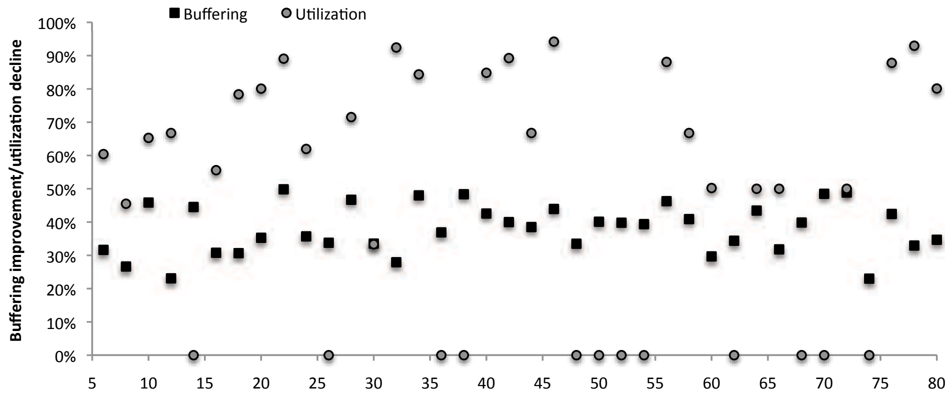


Figure 4.7: Impact of the LOP priority assignment on the buffering requirements and the processor utilization.

does not influence the achievable throughput. Compromised solutions can be obtained by excluding some precedences in the deadlines adjustment step.

Let us now address buffer minimization in fixed-priority scheduling. Figure 4.7 shows the impact of the LOP priority assignment on buffering and processor utilization compared to the DM priority assignment. The average improvement is equal to 37.99% with a low standard deviation (0.07). This is a good improvement that comes at the price of a throughput decline with an average equal to 48.31%. Thus, LOP priority assignment can be used for systems with strong memory constraints. We should also note that LOP assignment does not decrease the throughput of 31.57% of the graphs. Those graphs are graphs whose factor B is too large (compared to σ) so that changing the priorities does not affect the throughput.

In a second configuration, we investigate the constrained LOP priority assignment

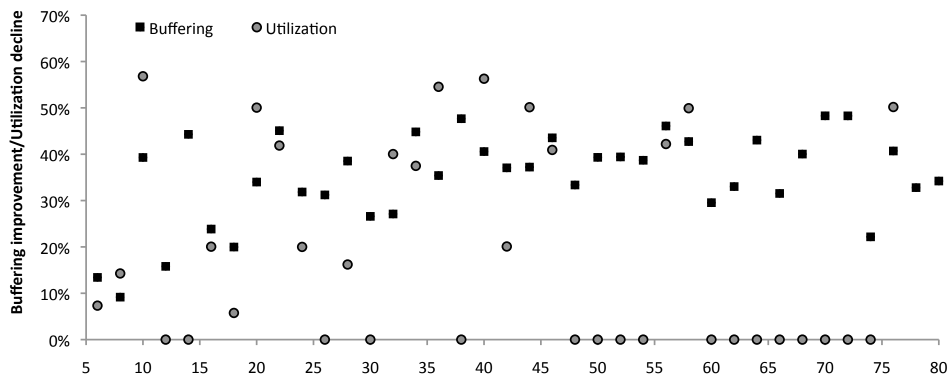


Figure 4.8: Impact of the constrained LOP priority assignment on the buffering requirements and the processor utilization.

which consists in constraining the precedence distance between the computed assignment and the DM priority assignment to not exceed a given threshold. Figure 4.8 shows the impact of the constrained LOP priority assignment on buffering and processor utilization for a threshold equal to $\frac{N(N-1)}{10}$. We obtained a slightly less buffering improvement (average equal to 34.99%) but for less throughput decrease (average equal to 21.83%). Furthermore, the constrained LOP assignment does not decrease the utilization in 44.73% of the cases.

In the third configuration, we would like to find a priority assignment that improves the buffering requirements without decreasing the processor utilization (compared to the DM utilization) by more than 15%. Therefore, we use the utilization distance instead of the precedence distance; and hence the heuristic presented in Algorithm 3 (p. 97). However, since the heuristic is based on a local enumeration, we do not expect to obtain a huge buffering improvement as the one obtained in the previous experiment (based on the ILP exact solution). We take $W = 7$ (for an exact solution, we must take $W = N$) and a threshold $u = 0.15$. Figure 4.9 represents the obtained results. The average improvement in buffering requirements is equal to 8.60% (with a standard deviation of 0.1). This is a meaningful improvement that comes at no price (a negligible average throughput decline of 1.13%). Furthermore, it is worth noticing that for $N > 32$, value $W = 7$ is not large enough to have a good enumeration.

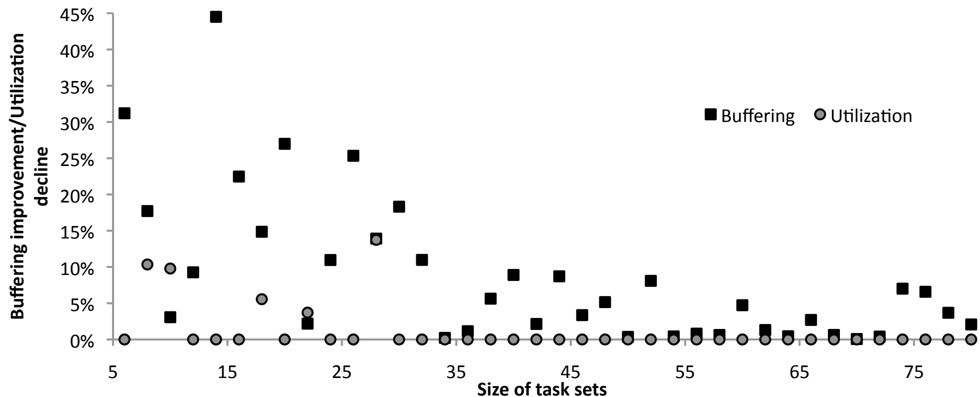


Figure 4.9: Impact of the priority assignment with utilization distance on the buffering requirements and the processor utilization.

4.2 Symbolic schedulability analysis

In this section, we will compare the complexity of our symbolic schedulability algorithms with that of basic enumerative solutions using a huge set of randomly generated task sets. This comparison will also demonstrate that our algorithms compute the same solutions as the basic enumerative algorithms. For a machine-independent comparison, we will compute in each time the number of checked absolute deadlines in EDF scheduling and the number of computed response times in FP scheduling. The task sets are

generated as follows:

1. Connected graphs: For each task p_i, we generate three parameters (C_i, α_i, β_i) such that $\pi_i = \alpha_i T$ and $d_i = \beta_i T$. The `UUniFast` algorithm [36] is used to generate uniformly distributed $\frac{C_i}{\alpha_i}$ values. Worst-case execution times are uniformly distributed in the interval $[100, 1000]$. Parameters α_i and β_i are uniformly generated by fixing the value of factor B and the value of an experimental parameter $D \in]0, 1]$ so that $\forall p_i : \beta_i \in [D\alpha_i, \alpha_i]$. We will show the obtained results for different configurations of N, B, and D.

1. Disconnected graphs: A disconnected graph consists of a set of connected components. Hence, we use the above mentioned method to generate $\mathcal{L}$ components and we then take the union of these components. Let N_j be the number of actors in the jth component (hence, $N = \sum_{j=1}^{\mathcal{L}} N_j$). We generate the components such that $\forall j : N_j = \frac{N}{\mathcal{L}}$; while factors B_j are uniformly distributed in a given interval.

4.2.1 EDF scheduling

Let us start with the `SQPA` algorithm (hence, connected graphs) to be compared with the following basic algorithm: starting from $k = 0$; test condition $h(t) \leq t$ at each absolute deadline $t \in [0, L(T_k)]$. If there is a deadline miss, then increment k. Repeat the same process until reaching the first feasible task set. Figure 4.10 shows the obtained results. Each point in the diagram is the ratio of the average number of checked deadlines by the enumerative solution (for 2000 task sets) to the average number of checked deadlines by `SQPA`. For each configuration, we denote by E the average number of checked points per task (obtained by the enumerative solution) to indicate the complexity of the problem.

The complexity of the symbolic schedulability problem increases inversely with factor B. Indeed, if B has a small value, then the number of checked deadlines will be large since each time a deadline is missed, T is increased only by a small quantity.

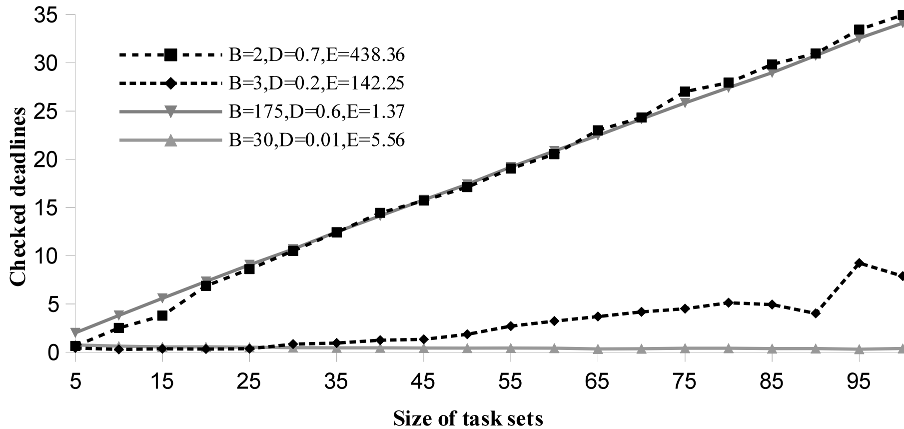


Figure 4.10: Performance of the `SQPA` algorithm.

For dataflow graphs with matched I/O rate channels, factor B is generally small, and the schedulability problem is hence more complicated. The SQPA algorithm outperforms the enumerative solution in all cases except cases with small N or very small D. When deadlines are too constrained, deadline misses could be detected earlier by a forward search than by a backward search. So, for dataflow graphs with few actors or highly constrained deadlines, it is better to use the enumerative solution than the SQPA algorithm.

In order to emphasis the impact of constrained deadlines on the performance of SQPA, we have conducted the following experiment. We fix $N = 60$ and $B = 2$, and use implicit deadlines for all tasks except for F tasks which will have constrained deadlines $d_i = \beta_i T$ such that $\beta_i \in [D\alpha_i, (D + 0.03)\alpha_i]$. The obtained results are shown in Figure 4.11. Each point in the diagram represents the average number of checked deadlines by either the enumerative solution or the SQPA algorithm for 2000 task sets.

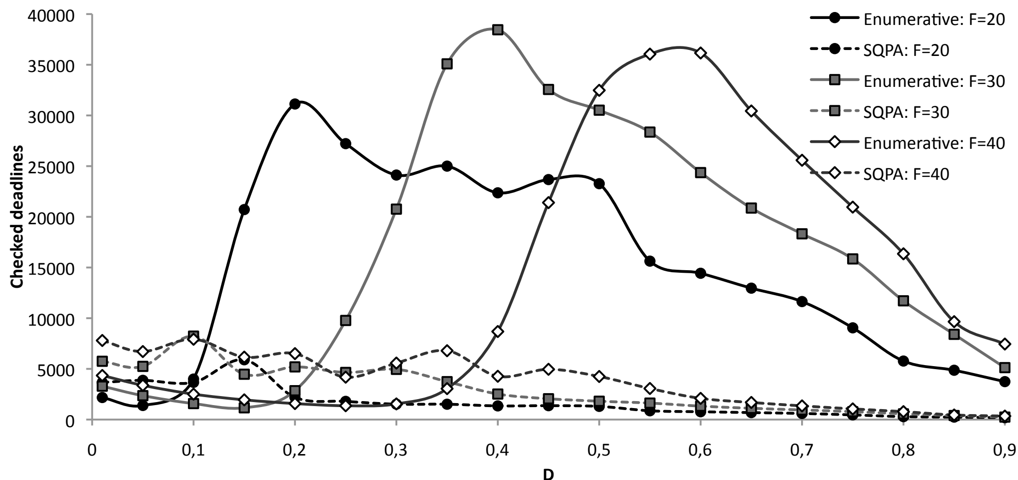


Figure 4.11: Impact of constrained deadlines on the performance of the SQPA algorithm.

This experiment shows that the complexity of the symbolic schedulability problem is lower for small and large values of D. This means that the schedulability problem is simpler when task sets have implicit deadlines or highly constrained deadlines. Furthermore, the enumerative solution outperforms SQPA only for task sets with highly constrained deadlines. The value of D after which SQPA outperforms the enumerative approach depends on the number of tasks with constrained deadlines (i.e. value of F).

The second algorithm to consider is the FFDBF−SQPA symbolic schedulability algorithm for connected graphs which will be compared with the following basic enumerative algorithm: Starting from $k = 0$ and for each value of $\gamma \in [\gamma_{min}(T_k), \gamma_{\max}(T_k)[$, test condition $h(t, \gamma) \leq t$ at each absolute deadline $t \in [0, L(T_k)]$. If a deadline miss occurs, then increment either γ (if $\gamma + \epsilon < \gamma_{\max}(T_k)$) or k. Repeat the same process until reaching the first feasible task set. Figure 4.12 represents the obtained results for $M = 2$. They are quite similar to that of SQPA; i.e. the FFDBF−SQPA algorithm outperforms

the enumerative approach for non-small values of N and D. We note that in case of FFDBF−SQPA, parameter D has more impact than factor B on the complexity of the problem.

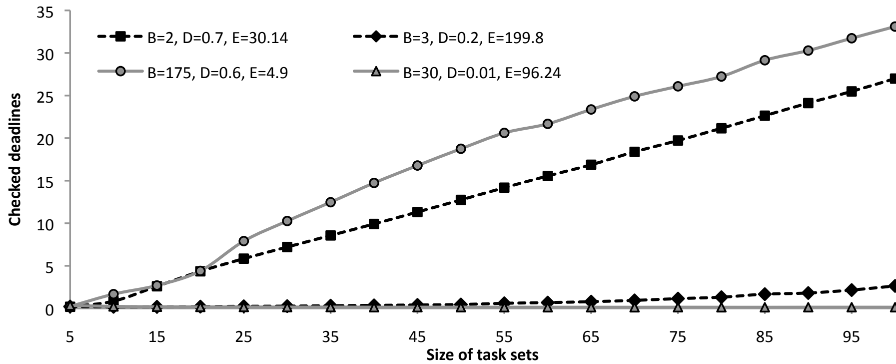


Figure 4.12: Performance of the FFDBF−SQPA algorithm.

Let us now measure the impact of the number of processors on the performance of the FFDBF−SQPA algorithm. So, we fix $N = 100$ and $B = 2$ while we vary M from 2 to 20. The obtained results for two values of D ($D = 0.6$ and $D = 0.2$) are shown in Figure 4.13. Each point in the diagram represents the average number of checked deadlines by either the enumerative solution or the FFDBF−SQPA algorithm for 2000 task sets. Firstly, you may notice the drastic impact of parameter D on the number of checked points (by comparing curves *Enumerative: D=0.6* and *Enumerative: D=0.2*). This impact is almost constant over the entire range of number of processors (a factor of 1.6). Regarding the impact of parameter M on the performance of FFDBF−SQPA, we found that the benefit (i.e. reduction of complexity in comparison with the enumerative

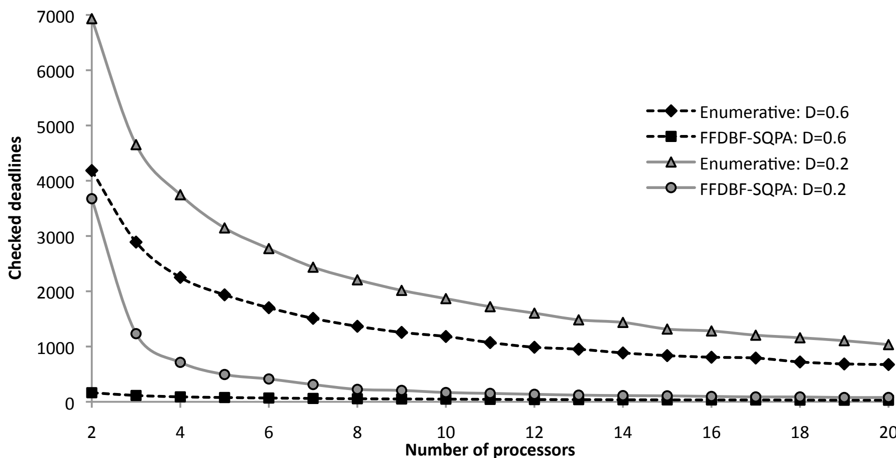


Figure 4.13: Impact of the number of processors on the performance of the FFDBF−SQPA algorithm.

solution) is almost constant for $D = 0.6$ and monotonic for $D = 0.2$. Furthermore, the complexity of the schedulability problem decreases when adding more processors.

The last algorithm to consider is the `DF-B&B SQPA` algorithm (i.e. `SQPA` for disconnected graphs). We generate disconnected graphs with $\mathcal{L} = 2$, $B_j \in [10, 20]$, and $T_j^u = 10\lceil\sigma_j\rceil$. The enumerative solution proceeds as follows: starting from $\mathcal{T} = \mathcal{T}^l$, test condition $h(t) \leq t$ at each absolute deadline in $[0, L(\mathcal{T})]$. If there is a miss, then put $\mathcal{T} =$ `getNext()` and repeat the same process until reaching the first feasible task set. Function `getNext()` returns the next unexplored point in the $\mathcal{T}$-space that results in the maximum utilization. The huge overhead of this method is not considered in the comparison. Figure 4.14 shows the obtained results. Each point in the diagram is the ratio of the average number of checked deadlines by the enumerative solution (for 100 task sets) to the average number of checked deadlines by `DF-B&B SQPA`. For each configuration, we denote by E the average number of checked points per task (obtained by the enumerative solution) to indicate the complexity of the problem. Again, the symbolic schedulability problem is more complex for large values of D. Furthermore, `DF-B&B SQPA` outperforms the enumerative solution for non-small values of N and D.

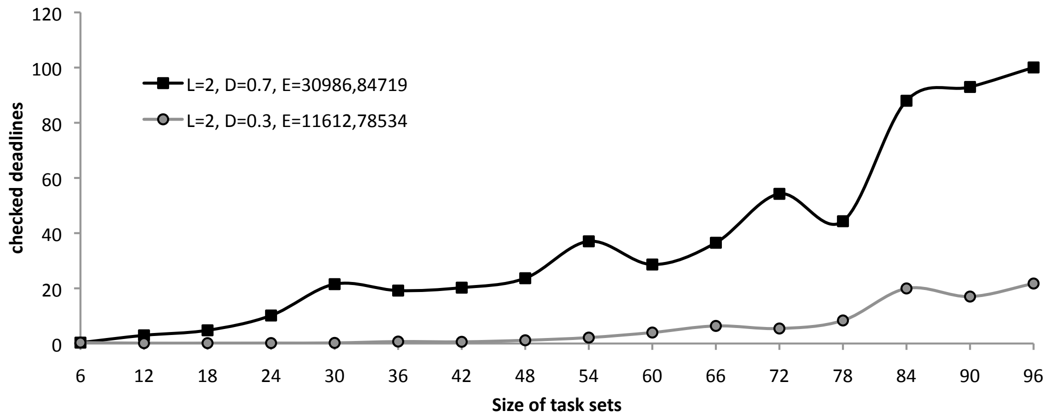


Figure 4.14: Performance of the `DF-B&B SQPA` algorithm.

4.2.2 Fixed-priority scheduling

Let us first consider the `SRTA` algorithm (i.e. connected dataflow graphs) with fixed priorities (i.e. a unique priority region). We assume that tasks have priorities in the same order in which they were randomly generated. The obtained results are shown in Figure 4.15 where each point represents the ratio of the average number of computed response times by the enumerative solution (for 2000 task sets) to the average number of computed response times by `SRTA`. The enumerative algorithm consists in the following steps: starting from $k = 0$, perform RTA. If the task set is not feasible, then increment k and repeat the same process until reaching the first feasible task set. The difference between the performance of `SRTA` and the performance of the basic algorithm is quite huge and unexpected. For instance, for $N = 100, B = 2$, and $D = 0.7$, `SRTA` requires

the computation of only 13 response times while the basic approach requires to compute more than 224426 response times. In fact, for the four configurations, SRTA requires to compute at most 13 response times.

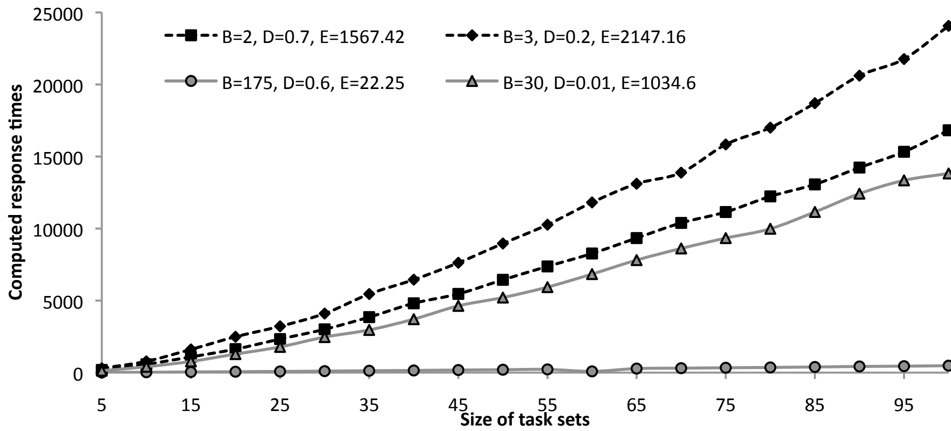


Figure 4.15: Performance of the SRTA algorithm.

The second schedulability algorithm to address is the DF−B&B SRTA algorithm (i.e. SRTA for disconnected graphs). We follow the same approach, described in the previous section, to generate disconnected graphs with $\mathcal{L} = 2$, $B \in [10, 20]$, and $T_j = 100\lceil\sigma_j\rceil$. The enumerative solution proceeds as follows: starting from $\mathcal{T} = \mathcal{T}^l$, perform RTA. If there is a deadline miss, then put $\mathcal{T} = \texttt{getNext}()$ and repeat the same process until reaching the first feasible task set. Figure 4.16 presents the obtained results where each point in the diagram represents the ratio of the average number of computed response times by the enumerative solution (for 50 task sets) to the average number of computed response times by DF−B&B SRTA. The performance of the DF−B&B SRTA algorithm is not as good as expected, especially in comparison with the performance of the SRTA algorithm. Thus, the problem should be further researched to identify better algorithms.

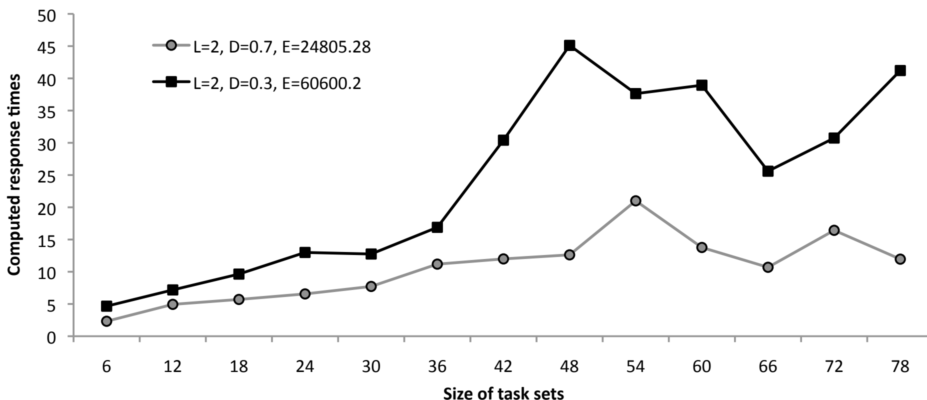


Figure 4.16: Performance of the DF−B&B SRTA algorithm.

4.3 Application: Design of SCJ/L1 systems

In recent years, there has been a growing interest in using Java for safety-critical systems such as flight control systems, railway signaling systems, robotic surgery machines, etc. The high productivity of Java compared to low-level languages is one among multiple reasons that encourage efforts to develop Java environments for both high-end and low-end embedded devices. FijiVM, JamaicaVM, PERC, PERC Pico, KESO, and Muvium are examples of such environments. Much ongoing effort, mainly made by the JSR-302 expert group, is focused on developing the Safety-Critical Java (SCJ) specification [104]. SCJ is a subset of Java augmented by the Real-Time Specification for Java (RTSJ) [90] and intended to develop applications that are amenable to certification under safety critical standards (such as DO-178B, Level A).

To better meet domain-specific safety requirements, the SCJ specification defines three levels of compliance (Levels 0, 1, and 2), each with a different model of concurrency, each aiming at applications of specific criticality. By the time of writing, there are few and incomplete implementations (only levels 0 and 1) of SCJ specification; for instance oSCJ [156] implemented on top of FijiVM [155], SCJ implemented on top of HVM [169], and the prototype implementation of SCJ on the Java processor JOP [162]. We have chosen Level 1 because it is less restricted than Level 0 and more analyzable than Level 2 not yet implemented. Indeed, SCJ/L0 supports only periodic handlers scheduled by a cyclic executive on a single processor; while SCJ/L2 has a much complicated concurrency model with nested missions, no-heap real-time threads, inter-processor job migration, self-suspension, etc.

In this section, we will focus on the concurrency model of SCJ/L1 rather than its memory model. SCJ/L1 relies entirely on periodic and aperiodic event handling. Each handler has its own server thread; this is however inconsistent with RTSJ in which a many-to-one relationship between handlers and servers is allowed [193]. Handlers communicate through shared memory; concurrency therefore becomes an issue as the programmer has to deal with data races, priority inversion, and deadlocks. In order to avoid data races, lock-based synchronization protocols are required though they are extremely complex and require pessimistic schedulability analyses.

Our objective is to propose a dataflow design model of SCJ/L1 applications (as an extension to the affine dataflow model) in which handlers communicate only through lock-free channels. Using dataflow diagrams to model real-time Java applications is not a new idea. Indeed, dataflow models of computation have been used in many real-time Java programming models such as Eventrons [170], Reflexes [171], Exotasks [6], StreamFlex [172], and Flexotasks [5]. The main goal of those restricted subsets of Java is to achieve lower latencies than what can be achieved by the real-time garbage collector. They attest the software engineering benefits of the dataflow model by offering some model-driven development capabilities such as automatic code generation. Some of them also provide type systems to ensure memory isolation; i.e. tasks can communicate only through specific mechanisms such as buffers and transactional memory. The above mentioned issues have been already partially solved by the SCJ memory model [55] or the SCJ annotation checker [186]. *Those dataflow programming models lack however the*

necessary tools for computing buffer sizes and priority-driven scheduling parameters.

4.3.1 Concurrency model of SCJ/L1

A SCJ/L1 application consists of a set of missions executed in sequence; nested missions are hence not allowed. The infrastructure `MissionSequencer` thread is responsible of creating and terminating missions, one after another. As depicted in Figure 4.17, a mission starts in an initialization phase during which schedulable objects (periodic and aperiodic event handlers) are created. Thus, the number and scheduling parameters (i.e. periods, phases, deadlines, priorities, etc.) of handlers are known at compile-time. Handlers are released during the mission execution phase such that periodic handler releases are time-triggered while aperiodic handler releases are event-triggered. SCJ/L1 specification does not support sporadic releases; therefore, aperiodic handlers can have only soft deadlines since the assumptions necessary to check their schedulability are not part of the profile. Each handler overrides the `handleAsyncEvent()` method to provide the computational logic executed when the handler is released. *This is somehow similar to firing functions of actors in dataflow graphs and hence justifies our choice of a dataflow design model.*

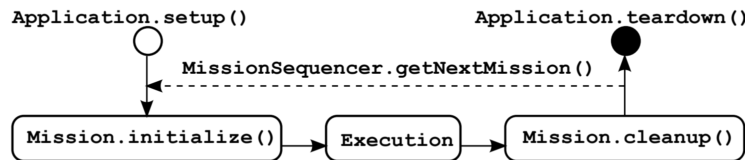


Figure 4.17: SCJ mission life cycle.

Periodic and aperiodic handlers are bound asynchronous event handlers; i.e. each handler is permanently bound to a single implementation thread. Each handler has a fixed priority; and the set of handlers is executed in parallel with a full preemptive priority-based scheduler. Handlers cannot self suspend and access to shared data is controlled by `synchronized` methods and statement blocks to avoid race conditions. The scheduler must implement the priority ceiling emulation protocol.

Scheduling aperiodic tasks based on their priorities may cause some lower priority hard tasks to miss their deadlines. Among the proposed solutions, aperiodic servers have been devised to improve the average response time of soft aperiodic tasks; examples of such techniques are: polling servers, deferrable servers, sporadic servers, priority exchange servers, etc. See Section 1.4.3 for more details. The SCJ specification is silent about using such approaches to execute aperiodic handlers although sporadic servers, for instance, are now supported by the POSIX standard P1003.1d. In the sequel, we propose to use sporadic servers; nevertheless, the presented technique can be easily adapted to consider polling servers or deferrable servers.

SCJ/L1 also supports multiprocessor implementations which require full partitioned scheduling (through the notion of Affinity Sets). Each handler can execute on a fixed processor with no migration (i.e. a partitioned FP scheduling).

4.3.2 Dataflow design model

Since missions are independent, each mission will be separately designed as a dataflow graph; while the mission sequencer can be modeled by a finite state machine. So, an actor in a dataflow graph represents a SCJ handler. The necessary user-provided information about a mission are illustrated in Figure 4.18.

Circle nodes represent periodic actors. The user must provide the worst-case execution time of each actor and the production and consumption rates of FIFO channels (solid arrows between periodic actors). Those rates can be just a safe abstraction of the actual production and consumption rates that can be obtained by some static code analysis. A FIFO channel `e` will be simply implemented as a cyclic array `E` with a fixed size `s` such that the instruction `e.set(v)` is implemented as `{E[i]=v;i=(i+1)%s;}` where `i` is a local index in the producer. Calls for the `get()` method are implemented in a similar way. The overflow and underflow analyses will ensure that there will be no need for synchronization protocols to access the arrays. Two non-communicating periodic actors can be linked together via an affine relation to relate their speeds to each other. Actors p_3 and p_2 are $(2, 0, 1)$-affine-related which means that actor p_2 is twice as fast as actor p_3.

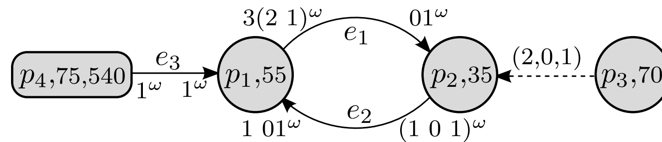


Figure 4.18: An ultimately cyclo-static dataflow graph with aperiodic actors.

Rectangle nodes represent aperiodic handlers. Actor p_4 is an aperiodic handler bound to a sporadic server thread which has a capacity equal to 75 and a replenishment period equal to 540. Parameters of servers that minimize the average response times are generally obtained by simulation. Some selection criteria have been proposed in [22]. Since there can be multiple servers in the system, a capacity sharing protocol like the one described in [23] may increase the responsiveness of aperiodic tasks.

Besides having unknown (or extremely large) worst-case execution times, aperiodic tasks have unknown interarrival time of requests. Therefore, an aperiodic task cannot communicate with other tasks via simple FIFO channels. Indeed, these communications may be unbounded or empty and hence block the other tasks for an indeterminate time. In our design model, an aperiodic task communicates with other (periodic or aperiodic) tasks through Cyclical Asynchronous Buffers (CAB) [61]. CABs offer bounded non-blocking communications. Tokens in a CAB are maintained until they are over-written by the producer. Hence, some produced tokens are lost if the producer is faster than the consumer; and the consumer may read the same tokens several times if it is faster than the producer. This is not a problem in many control applications, where tasks require fresh data rather than the complete stream. CABs were used in HARTIK system [52] and SimpleRTK. Sample-and-hold communication mechanisms are used in many design models; for instance in the Architecture Analysis and Design Language (AADL) [161]

and the Prelude compiler [149]. Based on the parameters of the aperiodic servers bound to the aperiodic tasks, it is possible to compute the average sizes of CABs.

The user may provide further optional information: (1) Lower bounds and upper bounds on periods. (2) Constrained deadlines of tasks as fractions (less than one) of periods; otherwise deadlines are assumed to be implicit. (3) The number of identical processors for multiprocessor implementations; otherwise uniprocessor scheduling is considered. (4) Buffer sizes and number of initial tokens in channels.

Implementation

The presented dataflow design model comes with a development tool integrated in the Eclipse IDE for easing the development of SCJ/L1 applications and enforcing the restrictions imposed by the design model. It consists of a GMF editor where applications are designed graphically and timing and buffering parameters can be synthesized. Indeed, affine scheduling is first applied on the dataflow subgraph that consists only of periodic actors. Then, symbolic FP schedulability analysis considers both periodic and aperiodic actors. Thanks to equations such as Equation 1.14 (p. 45), SRTA can be easily extended to consider also sporadic servers. Through a model-to-text transformation, using Acceleo, the SCJ code for missions, interfaces of handlers, and the mission sequencer is automatically generated in addition to the annotations needed by the memory checker.

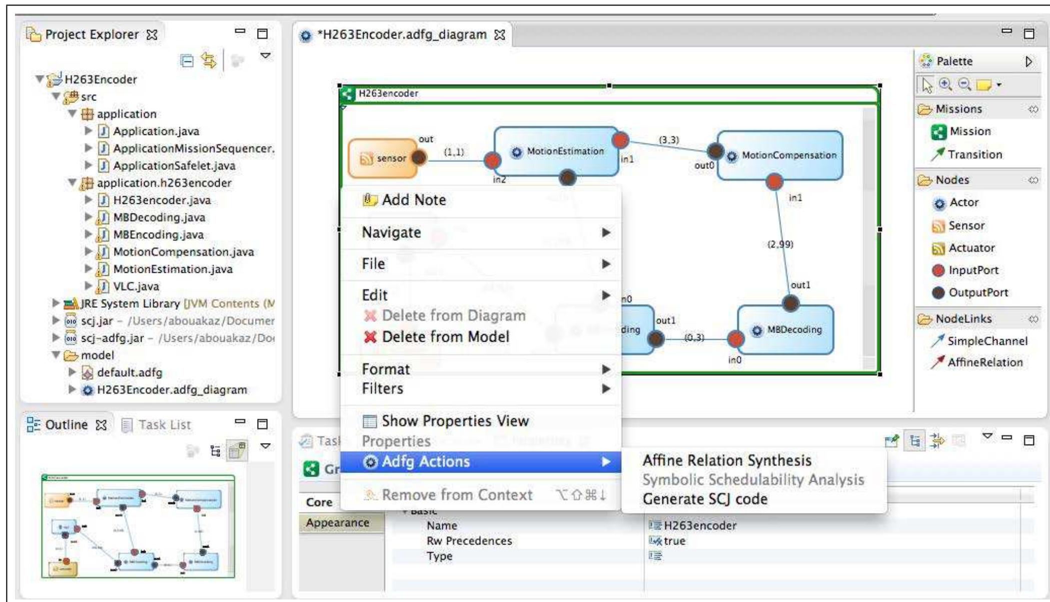


Figure 4.19: Graphical editor for SCJ/L1 applications design.

Channels are implemented as cyclic arrays or CABs and a fixed amount of memory is hence reused to store the infinite streams of tokens. Channels are instantiated in the mission memory since event handlers execute in private memory areas and can communicate only through mission memory (or the immortal memory). For instance,

when a handler reads a token from a channel, it simply copies the token from the mission memory to its private memory.

The user must provide the SCJ code of all the `handleAsyncEvent()` methods. We have integrated the SCJ memory checker in our tool so that potential dangling pointers can be highlighted at compile-time. A dangling pointer is a reference from an object allocated in a long-lived memory area (e.g. the immortal memory which is destroyed only at the end of the application) to an object allocated in a short-lived memory area (e.g. the private memory of a handler which is destroyed each time its firing function completes).

4.4 Conclusion

In this chapter, we have performed several experiments on either real-life benchmarks or randomly generated graphs and task sets. We have compared our tool with the DARTS tool and shown that our symbolic schedulability tests are generally more accurate. We have also shown the improvement on buffering requirements that can be achieved by the priority assignment and deadlines adjustment techniques. Furthermore, we have shown that our symbolic schedulability tests are much faster than the basic enumerative solutions in most cases. Finally, we have briefly present how to use the affine scheduling technique to automatically generate SCJ/L1 applications from a dataflow specification.

Conclusion

The design of real-time safety-critical applications is getting more and more complex due to the ever-increasing functional and nonfunctional requirements. This calls for new design flows that solve the specification, validation, and synthesis problems. Formal models must be used to describe the system, then formal analysis techniques explore the design space to assess feasibility and find the optimal design before the costly implementation phase. Finally, automatic synthesis techniques should be used as much as possible to generate correct by construction implementations.

Any design flow must ensure two key properties of a real-time safety-critical system, namely, functional determinism and temporal predictability. Dataflow models of computation (e.g. SDF, CSDF, etc.) are widely used to design stream-based embedded systems thanks to their inherent functional determinism. Since the introduction of the SDF model, a considerable effort has been made to solve the static-periodic scheduling problem; i.e. the construction of periodic infinite sequences of firings of actors that result in bounded, live, and complete executions. Ensuring boundedness and liveness is the essence of the proposed algorithms in addition to optimizing some nonfunctional performance metrics (e.g. buffer minimization, throughput maximization, code size minimization, etc.). We tried in Chapter 1 to present a survey of dataflow models of computation together with the existing algorithms for static-periodic scheduling of (C|H)SDF graphs.

Nowadays real-time embedded systems are so complex that real-time operating systems are used to manage hardware resources and host real-time tasks. Most of real-time operating systems rely on priority-driven scheduling algorithms (e.g. RM, EDF, etc.) instead of static-periodic schedules (also called cyclic executives) because of the inflexibility and difficult maintainability of cyclic executives. Real-time scheduling theory provides the necessary schedulability tests to verify that all real-time tasks will meet their deadlines even in the worst-case scenario. In Chapter 1, we have presented some of the schedulability tests regarding EDF and FP scheduling for uniprocessor and multiprocessor architectures. The underlying model of computation was the periodic task model which consists of a set of independent and concurrent real-time tasks.

This thesis has addressed the problem of implementing a dataflow specification as a set of independent real-time tasks to be executed on a system equipped with a real-time operating system [45, 43, 44]. This problem consists in mapping each actor in the dataflow graph to a periodic real-time task with appropriate scheduling parameters (i.e. period, first start time, priority, processor allocation, etc.). Certainly, properties

such as boundedness and liveness must also be considered. Since reads and writes in the periodic task model are non-blocking operations, we have talked more about overflow and underflow exceptions over communication channels. Real-time scheduling of dataflow graphs is not a trivial problem. We have solved it in two steps: abstract scheduling (Chapter 2) and symbolic schedulability analysis (Chapter 3). Let us recall the contributions made at each step:

Abstract scheduling

1. Activation-related schedules are presented, together with the necessary conditions for boundedness, liveness, and consistency, as a general framework to describe schedules of static dataflow graphs.

2. Affine relations are used to describe strictly periodic schedules. The objective of the ILP formalization of the problem was to minimize the buffering requirements in a machine-independent way.

3. We have also presented UCSDF (a generalization of CSDF), `FRStream` (a synchronous operational semantics for UCSDF graphs), multichannels, and how channels can share the same storage space.

Symbolic real-time schedulability analysis

1. We have created a new application field for parametric schedulability theory.

2. We have presented many symbolic EDF schedulability analyses (e.g. `SQPA`, `DF-B&B SQPA`, `FFDBF-SQPA`, `BF-EDF`, etc.) that aim at maximizing the processor utilization factor. We have shown that our schedulability tests have a lower complexity than the basic enumerative solutions. We have also presented a deadline adjustment technique (to encode scheduling precedences) that aims at minimizing the buffering requirements.

3. Finally, many symbolic FP schedulability tests are presented (e.g. `SRTA`, `DF-B&B SRTA`, etc.). We also presented some priority assignment strategies that aim at either minimizing the buffering requirements, maximizing the throughput, or finding a compromised solution.

In Chapter 4, we have shown that our tool (the ADFG tool) outperforms the only existing tool for real-time scheduling of dataflow graphs (the DARTS tool). We have also briefly presented our graphical editor for automatic synthesis of SCJ Level 1 applications from UCDF graph specifications. Certainly, the synthesis flow relies on affine scheduling and symbolic FP schedulability analysis to transform the dataflow specification to a set of independent periodic SCJ handlers.

Perspectives

This thesis is one among the fewest studies that have addressed the real-time scheduling problem of dataflow graphs. However, it offers by no means complete or optimal

solutions. Several points need to be researched further; for example:

1. *Expressive dataflow models:* Though the abstract scheduling approach (Section 2.2) has been presented for arbitrary rate functions and activation relations, we have only implemented a specific case where infinite integer sequences (used either to describe activation relations or rate functions) were ultimately periodic. It will be very interesting to find more expressive classes of integer sequences that could be used as rate functions and yet define an analyzable model.

2. *Symbolic schedulability analysis:* We have focused more on symbolic schedulability analysis of connected graphs. The design parameter space could be further relaxed to consider: disconnected graphs, arbitrary deadlines, more priority assignment strategies, etc. More performance metrics should also be addressed as energy consumption minimization. We believe that this research field is very large, complex, and not yet explored.

3. *Actor clustering:* In the presented scheduling approach, each actor is mapped to a periodic real-time task. However, a large set of real-time tasks could result in a noticeable run-time overhead (due to context switching, scheduling decisions, etc.). It may be hence reasonable to map many actors to a single real-time task so that each task consists in a static schedule of its actors.

4. *Design of SCJ\L1 applications:* To enhance functional determinism, we would like to develop an ownership type system, as the one presented in Reflexes, to ensure firstly that actors are strongly isolated; i.e. objects allocated within an actor are encapsulated unless they are full-immutable. This requirement ensures that the state of an actor cannot be altered by other actors. Typing rules such as "a non-private field must be either a final primitive or a final reference to a full-immutable object" and "non-private methods of an actor do not change its state or leak references to mutable objects" must be checked statically. Besides isolation of actors, the type system must ensure that actors can communicate only through buffers. Hence, "shared objects in the immortal or mission memory areas must be either final primitives or final references to full-immutable objects". Our future work will also consider to develop a static analysis that infers safe production and consumption rates from the user-provided SCJ code of firing functions and to connect our tool to existing worst-case execution time estimation tools.

Bibliography

[1] M. Adé, R. Lauwereins, and J. A. Peperstraete. Data memory minimization for synchronous data flow graphs emulated on DSP-FPGA targets. In *Proceedings of the 34th Annual Design Automation Conference*, pages 64–69, 1997.

[2] B. Anderson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 193–202, 2001.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[4] N. C. Audsley. On priority assignment in fixed priority scheduling. *Inf. Process. Lett.*, 79(1):39–44, 2001.

[5] J. Auerbach, D. F. Bacon, R. Guerraoui, J. H. Spring, and J. Vitek. Flexible task graphs: a unified restricted thread programming model for Java. *SIGPLAN Not.*, 43(7):1–11, 2008.

[6] J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. Rajan, H. Roeck, and R. Trummer. Java takes flight: time-portable real-time programming with Exotasks. *SIGPLAN Not.*, 42(7):51–62, 2007.

[7] F. Baccelli, G. Cohen, G. J. Oldsder, and J.-P. Quadrat. *Synchronization and linearity: an algebra for discrete event systems*. Wiley, 1992.

[8] T. P. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. In *Proceedings of the International Conference on Real-Time and Network Systems*, pages 119–130, 2006.

[9] M. Bamakhrama and T. Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the 9th ACM International Conference on Embedded Software*, pages 195–204, 2011.

[10] M. A. Bamakhrama and T. Stefanov. Managing latency in embedded streaming applications under hard-real-time scheduling. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 83–92, 2012.

[11] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. Implementation of speedup-optimal global EDF schedulability test. In *Proceedings of the 21st*

Euromicro Conference on Real-Time Systems, pages 259–268, 2009.

[12] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 321–329, 2005.

[13] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Trans. Comput.*, 55(7):918–923, 2006.

[14] S. K. Baruah, R. R. Howell, and L. E. Rosier. Feasibility problems for recurring tasks on one processor. *Theor. Comput. Sci.*, 118(1):3–20, 1993.

[15] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190, 1990.

[16] T. Basten and J. Hoogerbrugg. Efficient execution of process networks. In *Communicating Process Architectures*, pages 1–14, 2001.

[17] S. S. Battacharyya, E. A. Lee, and P. K. Murthy. *Software synthesis from dataflow graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[18] M. Benazouz, O. Marchetti, A. M. Kordon, and P. Urard. A new approach for minimizing buffer capacities with throughput constraint for embedded system design. In *Proceedings of the 8th ACS/IEEE Conference on Computer Systems and Applications*, pages 1–8, 2010.

[19] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In *Proceedings of the 10th International Conference on Concurrency Theory*, pages 162–177, 1999.

[20] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification & distributed code generation. *Information and Computation*, 163(1):125–171, 2000.

[21] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[22] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 68–78, 1999.

[23] G. Bernat and A. Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Syst.*, 22(1/2):49–75, 2002.

[24] G. Berry. The foundations of Esterel. In *Proof, language, and interaction*, pages 425–454. MIT Press, Cambridge, MA, USA, 2000.

[25] M. Bertogna and S. Baruah. Tests for global EDF schedulability analysis. *J. Syst. Archit.*, 57(5):487–497, 2011.

[26] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 149–160, 2007.

[27] M. Bertogna, M. Cirinei, and G. Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, pages 306–321, 2006.

[28] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling of DSP systems. *Trans. Sig. Proc.*, 49(10):2408–2421, 2001.

[29] S. Bhattacharyya, E. Deprettere, and B. Theelen. Dynamic dataflow graphs. In *Handbook of Signal Processing Systems*. Springer, 2nd edition, 2012.

[30] S. S. Bhattacharyya. *Compiling dataflow programs for digital signal processing*. PhD thesis, EECS Department, University of California, Berkeley, 1994.

[31] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee. Generating compact code from dataflow specifications of multirate signal processing algorithms. In *Readings in hardware/software co-design*, pages 452–464. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[32] S. S. Bhattacharyya and E. A. Lee. Looped schedules for dataflow descriptions of multirate signal processing algorithms. In *Journal of Formal Methods in System Design*, pages 183–205, 1994.

[33] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *J. VLSI Signal Process. Syst.*, 21(2):151–166, 1999.

[34] E. Bini. *The design domain of real-time systems*. PhD thesis, Scuola Superiore Sant'Anna, 2004.

[35] E. Bini and G. C. Buttazo. Schedulability analysis of periodic fixed priority systems. *IEEE Trans. Comput.*, 53(11):1462–1473, 2004.

[36] E. Bini and G. C. Buttazo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1-2):129–154, 2005.

[37] E. Bini, G. Buttazzo, and G. Buttazo. A hyperbolic bound for the rate monotonic algorithm. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 59–66, 2001.

[38] E. Bini and M. Di Natale. Optimal task rate selection in fixed priority systems. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 399–409, 2005.

[39] E. Bini, M. Di Natale, and G. Buttazzo. Sensitivity analysis for fixed-priority real-time systems. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 13–22, 2006.

[40] E. Bini, T. H. C. Nguyen, P. Richerd, and S. K. Baruah. A response-time bound in fixed-priority scheduling with arbitrary deadlines. *IEEE Trans. Comput.*, 58(2):279–286, 2009.

[41] G. Blisen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *Trans. Sig. Proc.*, 44(2):397–408, February 1996.

[42] A. Bonfietti, M. Lombardi, M. Milano, and L. Benini. Throughput constraint for synchronous data flow graphs. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 26–40, 2009.

[43] A. Bouakaz and J.-P. Talpin. Buffer minimization in earliest-deadline first scheduling of dataflow graphs. *SIGPLAN Not.*, 48(5):133–142, 2013.

[44] A. Bouakaz and J.-P. Talpin. Design of safety-critical Java level 1 applications using affine abstract clocks. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, pages 58–67, 2013.

[45] A. Bouakaz, J.-P. Talpin, and J. Vitek. Affine data-flow graphs for the synthesis of hard real-time applications. In *Proceedings of the 12th International Conference on Application of Concurrency to System Design*, pages 183–192, 2012.

[46] L. C. Briand and D. M. Roy. *Meeting deadlines in hard real-time systems : the rate monotonic approach*. IEEE, 1999.

[47] J. D. Brock and W. B. Ackerman. Scenarios: a model of non-determinate computation. In *Proceedings of the International Colloquim on Formalization of Programming Concepts*, pages 252–259, 1981.

[48] J. T. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, EECS Department, University of California, Berkeley, 1993.

[49] J. T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control systems. In *Proceedings of the 28th Annual Asilomar Conference on Signals, Systems, and Computers*, pages 508–513, 1994.

[50] A. Burns and S. Baruah. Sustainability in real-time scheduling. *Journal of Computer Science and Engineering*, 2(1):74–97, 2008.

[51] G. C. Buttazo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[52] G. Buttazzo and M. Di Natale. HARTIK: a hard real-time kernel for programming robot tasks with explicit time constraints and guaranteed execution. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 404–409, 1993.

[53] P. Caspi, C. Mazuet, and N. R. Paligot. About the design of distributed control systems: the quasi-synchronous approach. In *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security*, pages 215–226, 2001.

[54] P. Caspi and M. Pouzet. Synchronous Kahn networks. *SIGPLAN NOT.*, 31(6):226–238, 1996.

[55] A. Cavalcanti, A. Wellings, and J. Woodcock. The safety-critical Java memory model: a formal account. In *Proceedings of the 17th International Conference on Formal Methods*, pages 246–261, 2011.

[56] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analyzing properties in platfrom-based embedded system designs. In *Proceedings of the Conference*

on Design, Automation and Test in Europe, pages 190–195, 2003.

[57] C.-S. Chang. *Performance guarantees in communication networks*. Springer-Verlag, 2000.

[58] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Syst.*, 2(3):181–194, 1990.

[59] C. Chevalier and F. Pellegrini. PT-Scotch: a tool for efficient parallel graph ordering. *Parallel Comput.*, 34(6-8):318–331, 2008.

[60] A. Cimatti, L. Palopoli, and Y. Ramadian. Symbolic computation of schedulability regions using parametric timed automata. In *Proceedings of the 2008 Real-Time Systems Symposium*, pages 80–89, 2008.

[61] D. Clark. HIC: an operating system for hierarchies of servo loops. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 1004–1009, 1989.

[62] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principle of Programming languages*, pages 180–193, 2006.

[63] M. Cubric and P. Panangaden. Minimal memory schedules for dataflow networks. In *Proceedings of the 4th International Conference on Concurrency Theory*, pages 368–383, 1993.

[64] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):385–418, 2004.

[65] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 2002.

[66] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, 2011.

[67] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Trans. Comput.*, 57(9):1261–1276, 2008.

[68] R. de Groote, J. Kuper, H. Broersma, and G. J. M. Smit. Max-plus algebraic throughput analysis of synchronous dataflow graphs. In *Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 29–38, 2012.

[69] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. Yapi: application modeling for signal processing systems. In *Proceedings of the 37th Annual Design Automation Conference*, pages 402–405, 2000.

[70] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium: Proceedings, Colloque sur la programmation*, pages 362–376, 1974.

[71] U. C. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 23–30,

2003.

[72] B. P. Douglass. *Real-time agility: the harmony/ESW method for real-time and embedded systems development*. Addison-Wesley Profissional, 1st edition, 2009.

[73] F. Eisenbrand and T. Rothvoß. EDF-schedulability of synchronous periodic task systems is coNP-hard. In *Proceedings of the 21th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1029–1034, 2010.

[74] C. J. Fidge. Real-time scheduling theory. Technical report, Software Verification Center, School of Information Technology, The Univerity of Queensland, April 2002.

[75] N. Fisher, S. Baruah, and T. P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 118–127, 2006.

[76] P. Fradet, A. Girault, and P. Poplavkoy. SPDF: a schedulable parametric dataflow MoC. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 769–774, 2012.

[77] L. Fribourg, R. Soulat, D. Lesens, and P. Moro. Robustness analysis for scheduling problems using the Inverse Method. In *Proceedings of the 19th International Symposium on Temporal Representation and Reasoning*, pages 73–80, 2012.

[78] A. Gamatié. *Designing embedded systems with the Signal programming language: synchronous, reactive specification*. Springer Publisher Company, Inc., 2009.

[79] A. Gamatié. Design of streaming applications on MPSoCs using abstract clocks. In *Design, Automation and Test in Europe Conference*, pages 763–768, 2012.

[80] M. Geilen. Reduction techniques for synchronous dataflow graphs. In *Proceedings of the 46th Annual Design Automation Conference*, pages 911–916, 2009.

[81] M. Geilen and T. Basten. Requirements on the execution of Kahn process networks. In *Proceedings of the 12th European Conference on Programming*, pages 319–334, 2003.

[82] M. Geilen, T. Basten, and S. Stuijk. Minimizing buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 42nd Annual Design Automation Conference*, pages 819–824, 2005.

[83] A. H. Ghamarian, M. C. W. Geilen, T. Basten, and S. Stuikj. Parametric throughput analysis of synchronous data flow graphs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 116–121, 2008.

[84] A. H. Ghamarian, M. C. W. Geilen, T. Basten, B. D. Theelen, M. R. Mousavi, and S. Stuikj. Liveness and boundedness of synchronous dataflow graphs. In *Proceedings of the Formal Methods in Computer Aided Design*, pages 68–75, 2006.

[85] A. H. Ghamarian, M. C. W. Geilen, S. Stuikj, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij. Throughput analysis of synchronous data flow graphs. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design*, pages 25–36, 2006.

[86] A. H. Ghamarian, S. Stuikj, T. Basten, M. C. W. Geilen, and B. D. Theelen. Latency minimization for synchronous data flow graphs. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 189–196, 2007.

[87] S. Goddard. Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, pages 60–71, 1997.

[88] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25(2-3):187–205, 2003.

[89] J. Goossens and P. Richard. Performance optimization for hard real-time fixed priority tasks. In *Proceedings of the 12th International Conference on Real-Time Systems*, 2004.

[90] J. Gosling and G. Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[91] Z. Gu, M. Yuan, N. Guan, M. Lv, X. He, Q. Deng, and G. Yu. Static scheduling and software synthesis for dataflow graphs with symbolic model-checking. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 353–364, 2007.

[92] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. POLYCHRONY for system design. *Journal of Circuits, Systems, and Computers*, 12(3):261–304, 2003.

[93] S. Ha and H. Oh. Decidable dataflow models for signal processing: synchronous dataflow and its extensions. In *Handbook of Signal Processing Systems*. Springer, 2nd edition, 2012.

[94] N. Halbwachs. *Synchronous programming of reactive systems*. Springer-Verlag, Berlin, Heidelberg, 1993.

[95] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[96] N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design*, pages 3–14, 2006.

[97] T. A. Henzinger and J. Sifakis. The embedded systems challenge. In *Proceedings of the 14th International Conference on Formal Methods*, pages 1–15, 2006.

[98] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185, 1974.

[99] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, March 2005.

[100] K. Jeffay and D. Bennett. A rate-based execution abstraction for multimedia computing. In *Proceedings of the 5th International Workshop on Network and Operating*

System Support for Digital Audio and Video, pages 64–75, 1995.

[101] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, pages 129–139, 1991.

[102] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[103] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.

[104] JSR-302. *Safety critical Java technology specification*, 2010.

[105] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[106] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.

[107] D. J. Kaplan. An introduction to the processing graph method. In *Proceedings of the 1997 International conference on Engineering of Computer-Based Systems*, pages 46–52, 1997.

[108] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.

[109] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal of Applied Mathematics*, 14(6):1390–1411, November 1966.

[110] T. Kavitha, C. Liebchen, K. Mehlhorn, D. Michail, R. Rizzi, T. Ueckerdt, and K. A. Zweig. Cycle bases in graphs characterization, algorithms, complexity, and applications. *Computer Science Review*, 3(4):199–243, 2009.

[111] J. C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, pages 547–550, 2002.

[112] T. T. H. Le, L. Palopoli, R. Passerone, and Y. Ramadian. Timed-automata based schedulability analysis for distributed firm real-time systems: a case study. *International Journal on Software Tools for Technology Transfer*, 2012.

[113] E. A. Lee. Computing for embedded systems. In *IEEE Instrumentation and Measurement Technology Conference*, pages 1830–1837, 2001.

[114] E. A. Lee and S. Ha. Scheduling trategies for multiprocessor real-time DSP. In *IEEE Global Telecommunication Conference and Exhibition. Communication Technology for the 1990s and Beyond*, pages 1279–1283, 1989.

[115] E. A. Lee and E. Matsikoudis. The semantics of dataflow with firing. In G. Huet, G. Plotkin, J.-J. Lévy, and Y. Bertot, editors, *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, chapter 4, pages 71–94. Cambridge University Press, 1 edition, 2009.

[116] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow

programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[117] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, pages 1235–1245, 1987.

[118] E. A. Lee and T. M. Parks. Dataflow process networks. In G. De Micheli, R. Ernst, and W. Wolf, editors, *Readings in hardware/software co-design*, pages 59–85. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[119] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 17(12):1217–1229, 2006.

[120] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of the 10th Real Time Systems Symposium*, pages 166–171, 1989.

[121] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[122] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[123] J. Y. T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Inf. Process. Lett.*, 11(3):115–118, 1980.

[124] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[125] Q. Li and C. Yao. *Real-time concepts for embedded systems*. CMP Books, 2003.

[126] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput.*, 44(12):1429–1442, 1995.

[127] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[128] N. Liveris, C. Lin, J. Wang, H. Zhou, and P. Banerjee. Retiming for synchronous data flow graphs. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pages 480–485, 2007.

[129] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Syst.*, 28(1):39–68, 2004.

[130] W.-c. Lu, J.-W. Hsieh, W.-K. Shih, and T.-W. Kuo. A faster exact schedulability analysis for fixed-priority scheduling. *J. Syst. Softw.*, 79(12):1744–1753, 2006.

[131] J. W. S. Lui. *Real-time systems*. Prentice-Hall, 2000.

[132] W. Lui, Z. Gu, J. Xu, Y. Wang, and M. Yuan. An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 61–70, 2009.

[133] N. Lynch and E. W. Stark. A proof of the Kahn principle for input/output

automata. *Inf. Comput.*, 82(1):81–92, July 1989.

[134] L. Mandel, F. Plateau, and M. Pouzet. Lucy-n : a n-Synchronous Extension of Lustre. In *Proceedings of the 10th International Conference on Mathematics of Program Construction*, pages 288–309, 2010.

[135] R. Martí and G. Reinelt. *The linear ordering problem. Exact and heuristic methods in combinatorial optimization*. Berlin: Springer, 2011.

[136] A. J. Martin. The probe: an addition to communication primitives. *Inf. Process. Lett.*, 20:125–130, 1985.

[137] A. Masrur, S. Drössler, and G. Färber. Improvements in polynomial-time feasibility testing for EDF. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1033–1038, 2008.

[138] A. K. Mok. *Fundamental design problems of distributed systems for the hard real-time environment*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1983.

[139] O. Moreira, J.-D. Mol, M. Bekooij, and J. v. Meerbergeb. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 332–341, 2005.

[140] T. Murata. Circuit theoretic analysis and synthesis of marked graphs. *IEEE Trans. on Circuits and Systems*, pages 400–405, 1977.

[141] P. K. Murthy. *Scheduling techniques for synchronous and multidimentional synchronous dataflow*. PhD thesis, EECS Department, University of California, Berkeley, 1996.

[142] P. K. Murthy and S. S. Bhattacharyya. Shared memory implementations of synchronous dataflow specifications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 404–410, 2000.

[143] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee. Joint minimization of code and data for synchronous dataflow programs. *Form. Methods Syst. Des.*, 11(1):41–70, 1997.

[144] D.-I. Oh and T. P. Baker. Utilization bounds for N-processor rate monotonic scheduling with static processor assignment. *Real-Time Syst.*, 15(2):183–192, 1998.

[145] H. Oh, N. Dutt, and S. Ha. Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In *Proceedings of the 2006 Conference on Asia South Pacific Design Automation*, pages 497–502, 2006.

[146] H. Oh and S. Ha. Fractional rate dataflow model and efficient code synthesis for multimedia applications. *SIGPLAN NOT.*, 37(7):12–17, 2002.

[147] H. Oh and S. Ha. Memory-optimized software synthesis from dataflow program graphs with large size data samples. *EURASIP J. Appl. Signal Process.*, 2003:514–529, 2003.

[148] T. W. O'Neil and E. H. M. Sha. Retiming synchronous data-flow graphs to reduce

execution time. *Trans. Sig. Proc.*, 49(10):2397–2407, 2001.

[149] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete event dynamic systems*, 21(3):307–338, 2011.

[150] T. M. Parks. *Bounded scheduling of process networks*. PhD thesis, EECS Department, University of California, Berkeley, 1995.

[151] T. M. Parks and E. A. Lee. Non-preemptive real-time scheduling of dataflow systems. In *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, pages 3235–3238, 1995.

[152] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cycle-static dataflow. In *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set)*, pages 204–210, 1995.

[153] K. Pingali and Arvind. Efficient demand-driven evaluation. part 1. *ACM trans . Program. Lang . Syst .*, 7(2):311–333, April 1985.

[154] K. Pingali and Arvind. Efficient demand-driven evaluation. part 2. *ACM trans . Program. Lang . Syst .*, 8(1):109–139, January 1986.

[155] F. Pizlo, L. Ziarek, and J. Vitek. Real time Java on resource-constrained platforms with Fiji VM. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 110–119, 2009.

[156] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing safety-critical Java applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 95–101, 2010.

[157] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Form. Methods Syst. Des.*, 28(2):111–130, 2006.

[158] Y. Ramadian. *Parametric real-time system feasibility analysis using parametric timed automata*. PhD thesis, Uiversità Degli Studi di Terento, 2012.

[159] I. Ripoll, A. Crespo, and A. K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Syst.*, 11(1):19–39, 1996.

[160] M. P. S. Ritz and H. Meyr. High level software synthesis for signal processing systems. In *Proceedings of the International Conference on Application-Specific Processors*, pages 679–693, 1992.

[161] SAE Aerospace (Society of Automative Engineers). Aerospace standard AS5506A: Architecture analysis and design language (AADL). *SAE AS5506A*, 2009.

[162] M. Schoeberl and J. R. Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 54–61, 2012.

[163] D. Seto, J. P. Lehoczky, and L. Sha. Task period selection and schedulability in real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages

188–198, 1998.

[164] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. On task schedulability in real-time control systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 13–21, 1996.

[165] L. Sha, T. Abdelzaher, K.-E. Arzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: a historical perspective. *Real-Time Syst.*, 28(2-3):101–155, 2004.

[166] M. Sjödin and H. Hansson. Improved response-time analysis calculations. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 399–408, 1998.

[167] I. Smarandache and P. Le Guernic. A canonical form for affine relations in SIGNAL. Technical report, INRIA, 1997.

[168] I. M. Smarandache, T. Gautier, and P. L. Guernic. Validation of mixed signal-alpha real-time systems through affine calculus on clock synchronisation constraints. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, volume 2, pages 1364–1383, 1999.

[169] H. Søndergaard, S. E. Korsholm, and A. P. Ravn. Safety-critical Java for low-end embedded platforms. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 44–53, 2012.

[170] D. Spoonhower, J. Auerbach, D. F. Bacon, P. Cheng, and D. Grove. Eventrons: a safe programming construct for high-frequency hard real-time applications. *SIGPLAN Not.*, 41(6):283–294, 2006.

[171] J. H. Spring, F. Pizlo, R. Guerraoui, and J. Vitek. Reflexes: abstractions for higly responsive systems. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 191–201, 2007.

[172] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: high-throuput stream programming in Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 211–228, 2007.

[173] B. Sprunt, J. P. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 251–258, 1988.

[174] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[175] S. Sriram and S. S. Bhattacharyya. *Embedded multiprocessors: scheduling and synchronization*. Marcel Dekker, Inc., New York, NY, USA, 2000.

[176] S. Sriram and E. A. Lee. Determining the order of processor transactions in statically scheduled multiprocessors. *J. VLSI Signal Process. Syst.*, 15(3):207–220, 1997.

[177] E. W. Stark. Concurrent transition system semantics of process networks. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming*

Languages, pages 199–210, 1987.

[178] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.

[179] S. Stuijk. *Predictable mapping of streaming applications on multiprocessors*. PhD thesis, Eindhoven University of Technology, 2007.

[180] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the 43rd Annual Design Automation Conference*, pages 899–904, 2006.

[181] S. Stuijk, M. Geilen, B. D. Theelen, and T. Basten. Scenario-aware dataflow: modeling, analysis, and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 404–411, 2011.

[182] S. Stuijk, M. C. W. Geilen, and T. Basten. SDF^3: SDF for free. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design*, pages 276–278, 2006.

[183] S. Stuikj, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th Annual Design Automation Conference*, pages 777–782, 2007.

[184] S. Stuikj, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Trans. Comput.*, 57(10):1331–1345, 2008.

[185] Y. Sun, R. Soulat, G. Lipari, É. André, and L. Fribourg. Parametric schedulability analysis of fixed priority real-time distributed systems. *CoRR*, 2013.

[186] D. Tang, A. Plsek, and J. Vitek. Static checking of safety critical Java annotations. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 148–154, 2010.

[187] E. Teruel, P. Chrzastowski-Wachtel, J. M. Colom, and M. Silva. On weighted T-systems. In *Proceedings of the 13th International Conference on Application and Theory of Petri Nets*, pages 348–367, 1992.

[188] B. D. Theelen, M. Geilen, T. Basten, J. Voeten, S. V. Gheorghita, and S. Stuikj. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *4th International Conference on Formal Methods and Models for Co-Design*, pages 185–194, 2006.

[189] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the International Symposium on Circuits and Systems*, pages 101–104, 2000.

[190] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages

365–376, 2010.

[191] S. R. V. Zivojnovic and H. Meyr. Optimizing DSP programs using multirate retiming transformation. In *Proceedings of EUSIPCO Signal Processing*, 1994.

[192] J. E. Vuillemin. On circuits and numbers. *IEEE Trans. Comput.*, 43:868–879, 1994.

[193] A. Wellings and M. Kim. Asynchronous event handling and safety critical Java. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 53–62, 2010.

[194] M. Wiggers, M. Bekooij, P. Jansen, and G. Smit. Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 10–15, 2006.

[195] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Proceedings of the 44th Annual Design Automation Conference*, pages 658–663, 2007.

[196] F. Zhang and A. Burns. improvement to quick processor-demand analysis for EDF-scheduled real-time systems. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 76–86, 2009.

[197] F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transactions on Computers*, 58:1250–1258, 2009.

[198] X.-Y. Zhu, T. Basten, M. Geilen, and S. Stuikj. Efficient retiming of multirate DSP algorithms. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 31(6):831–844, 2012.

[199] X.-Y. Zhu, M. Geilen, T. Basten, and S. Stuikj. Static rate-optimal scheduling of multirate DSP algorithms via retiming and unfolding. In *Proceedings of 18th Real Time and Embedded Technology and Applications Symposium*, pages 109–118, 2012.

Appendix A

Sets, orders, and sequences

This appendix introduces some basic mathematical concepts about ordered sets used in describing the different dataflow models of computation. Most of definitions are taken from the wonderful book of Davey and Priestly [65]. Furthermore, the appendix presents the sequence domain and its properties.

A.1 Ordered sets

Definition A.1 (Poset)**.** A partially ordered set (*poset* for short) is a pair $(A, \sqsubseteq)$ such that A is a set and $\sqsubseteq$ is a partial order relation on A (i.e. $\sqsubseteq$ is reflexive, antisymmetric, and transitive).

Definition A.2 (Maximal elements, the maximum)**.** Let B be a subset of a poset $(A, \sqsubseteq)$. Then $a \in B$ is a maximal element of B if $a \sqsubseteq b$ and $b \in B$ imply $a = b$. The set of maximal elements of B is denoted by $\mathrm{Max}(B)$. The maximum element of B, if any, is $\max(B) \in B$ such that $\forall a \in B : a \sqsubseteq \max(B)$.

Dually, we can define the minimal elements and the minimum of a subset. The maximum element of the poset, if any, is called the top element (denoted by $\top_A$); while the minimum element, if any, is called the bottom element (denoted by $\bot_A$).

Definition A.3 (Upper bounds, the least upper bound)**.** Let B be a subset of a poset $(A, \sqsubseteq)$. The set of upper bounds of B is $B^u = \{a \in A | \forall b \in B : b \sqsubseteq a\}$. The least upper bound of B, if any, is the minimum of B^u and it is denoted by $\bigsqcup B$.

Dually, we can define the set of lower bounds B^l and the greatest lower bound $\bigsqcap B$ of a subset B.

Definition A.4 (Join semi-lattice)**.** A join semi-lattice $(A, \sqsubseteq, \sqcup)$ is a poset $(A, \sqsubseteq)$ such that any two elements $a, b \in A$ have a least upper bound $a \sqcup b$ (i.e. $\bigsqcup\{a, b\}$).

Dually, we can define a meet semi-lattice. A lattice $(A, \sqsubseteq, \sqcup, \sqcap)$ is both a join semi-lattice and a meet semi-lattice.

Definition A.5 (complete lattice). A complete lattice $(A, \sqsubseteq, \bigsqcup, \bigsqcap)$ is a poset $(A, \sqsubseteq)$ such that any subset $B \subseteq A$ has a least upper bound $\bigsqcup B$ and a greatest lower bound $\bigsqcap B$ in A.

Definition A.6 (Ascending chain condition). A chain B of a poset $(A, \sqsubseteq)$ is a subset $B \subseteq A$ such that $\forall a, b \in B : (a \sqsubseteq b) \vee (b \sqsubseteq a)$. The poset satisfies the ascending chain condition (ACC) if and only if any infinite sequence $a_0 \sqsubseteq a_1 \sqsubseteq \ldots \sqsubseteq a_n \sqsubseteq \ldots$ of elements of A is not strictly increasing, that is $\exists k \geq 0 : \forall j \geq k : a_k = a_j$.

Definition A.7 (CPO). A complete partial order (CPO for short) $(A, \sqsubseteq, \bot_A)$ is a poset $(A, \sqsubseteq)$ with a bottom $\bot_A$ such that any increasing chain of A has a lower upper bound in A.

A.2 Fixed point theory

Definition A.8 (Monotone function). Let $f \in A \to A$ be a (total) function on a poset $(A, \sqsubseteq, \bigsqcup, \bigsqcap)$. f is monotone if and only if $\forall a, b \in A : a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$.

Definition A.9 (Fixed points). Consider $f \in A \to A$ with $(A, \sqsubseteq)$ a poset, an element $a \in A$ is (1) a fixed point if and only if $f(a) = a$; (2) the greatest fixed point if and only if $f(a) = a$ and $\forall b : f(b) = b \Rightarrow b \sqsubseteq a$; (3) the least fixed point if and only $f(a) = a$ and $\forall b : f(b) = b \Rightarrow a \sqsubseteq b$.

Theorem A.1 (Knaster-Tarski). A monotone function $f \in A \to A$, with $(A, \sqsubseteq)$ a complete lattice, has a least fixed point $\text{lfp}(f) = \bigsqcap\{a \in A | f(a) \sqsubseteq a\}$.

Definition A.10 (continuous function). Let $(A, \sqsubseteq, \sqcup)$ be a CPO. A function $f \in A \to A$ is continuous if and only if for every chain $B \subseteq A$, we have that $\bigsqcup f(B) = f(\bigsqcup B)$.

Lemma A.1 (CPO of continuous functions). For CPOs $(A, \sqsubseteq_1)$ and $(B, \sqsubseteq_2)$, let $[A \to B]$ be the set of all continuous total functions from A to B. $([A \to B], \sqsubseteq)$ is a CPO where $\forall f, g \in [A \to B]$, we have that $f \sqsubseteq g \Leftrightarrow \forall a \in A : f(a) \sqsubseteq_2 g(a)$.

Theorem A.2 (fixed point theorem for a continuous function on a CPO). Let $f \in A \to A$ be a continuous function on a CPO $(A, \sqsubseteq, \bot_A)$. The least fixed point exists and equals $\bigsqcup_{n \geq 0} f^n(\bot_A)$ such that $f^{n+1} = f^n \circ f$.

Theorem A.3 (fixed point theorem for a monotone function on a CPO). Every monotone function on a CPO has a least fixed point.

A.3 Sequences

Let A^∞ be the set of finite or denumerably infinite sequences over a set A. A finite sequence $s \in A^*$ is a total function $s : \{1, \ldots, n\} \to A$ for some natural number n such that $s(i)$ is the ith element of the sequence. The number n is the length of the sequence, denoted by $|s|$. The empty sequence, which has a length equal to 0, is denoted by ϵ. The

sequence $s : \mathbb{N}_{>0} \to A$ is an infinite sequence of length equals ω where $\forall i \in \mathbb{N} : i < \omega$. The set of infinite sequences is denoted by A^ω. Thus, $A^\infty = A^* \cup A^\omega$.

We often list the elements of a sequence s, writing $a_1\ a_2\ \dots\ a_{|s|}$ if $s \in A^*$, and $a_1\ a_2\ \dots$ if $s \in A^\omega$. For two sequences s_1 and s_2, the sequence $s_1.s_2$ represents their concatenation.

Definition A.11 (Prefix order)**.** A sequence s_1 is a *prefix* of a sequence s_2, and we write $s_1 \sqsubseteq s_2$, if and only if it exists s_3 such that $s_2 = s_1.s_3$. We say then that s_1 provides an approximate information about s_2. The relation $\sqsubseteq$ is a partial order on A^∞.

$(A^\infty, \sqsubseteq)$ is a CPO. Indeed, the poset $(A^\infty, \sqsubseteq)$ has a bottom element $\bot = \epsilon$. Furthermore, the lower upper bound of any finite non-empty chain is the greatest element, while the lower upper bound of any infinite chain $s_0 \sqsubseteq s_1 \sqsubseteq \dots \sqsubseteq s_n \sqsubseteq \dots$ is the infinite sequence s where $s(i)$ equals $s_k(i)$ for any k such that $|s_k| \geq i$.

$(A^\infty, \sqsubseteq)$ is a meet semi-lattice since the greatest lower bound of any two sequences is their longest common prefix. However, it is not a join semi-lattice (e.g. $a_1\ a_2$ and $a_2\ a_1$ do not have a lower upper bound).

A.3.1 Periodic sequences

An infinite constant sequence $s = a^\omega$ where $a \in A$ is a sequence such that $\forall i \in \mathbb{N}_{>0} : s(i) = a$. A periodic infinite sequence $s = v^\omega$, where v is a finite non-empty sequence, is a sequence such that $\forall i \in \mathbb{N}_{>0} : s(i) = v(1 + (i-1) \mod |v|)$. An ultimately periodic sequence $s = uv^\omega$, where u is a (possibly empty) finite sequence, is the concatenation of a periodic sequence v^ω to a finite sequence u. Hence, $\forall i \in \mathbb{N}_{>0} : s(i) = u(i)$ if $i \leq |u|$; and $s(i) = v(1+(i-|u|-1) \mod |v|)$ otherwise. So, every constant sequence is periodic ($|v| = 1$) and every periodic sequence is ultimately periodic ($|u| = 0$).

A.4 Tuples

An N-tuple $S \in A_1^\infty \times A_2^\infty \times \dots \times A_N^\infty$ is a collection of N sequences $s_i \in A_i^\infty$. We often list the sequences within a N-tuple, writing $[s_1, s_2, \dots, s_N]$. Thus, $S(i)$ represents the ith sequence of the tuple. For the sake of conciseness, the Cartesian product $A_1^\infty \times A_2^\infty \times \dots \times A_N^\infty$ is denoted by $\mathcal{A}^N$.

We extend the prefix order to $\mathcal{A}^N$ as the point-wise prefix order; i.e. if $S_1, S_2 \in \mathcal{A}^N$, then $S_1 \sqsubseteq S_2$ if and only if $\forall i : S_1(i) \sqsubseteq_i S_2(i)$. Since any direct product of CPOs is a CPO, then $(\mathcal{A}^N, \sqsubseteq, \bot)$ is a CPO such that $\bot = [\epsilon_1, \dots, \epsilon_N]$.

A.5 Real sequences

Arithmetic operations are extended to real sequences as follows. If s and r are two infinite sequences or finite sequences with $|s| = |r|$, then $\forall i : (s\ \mathtt{op}\ r)(i) = s(i)\ \mathtt{op}\ r(i)$. Furthermore, $s \leq r$ if and only if $\forall i : s(i) \leq r(i)$.

Definition A.12 (Cumulative functions)**.** The cumulative function of a real sequence $s \in \mathbb{R}^\infty$ is the sequence $\oplus s$ such that $\forall i \in \{1, \ldots, |s|\} : \oplus s(i) = \sum_{j=1}^{i} s(j)$. By convention, $\oplus s(0) = 0$.

The sum of all elements of a finite sequence $v \in \mathbb{R}^*$, denoted by $\|v\|$, is equal to $\oplus v(|v|)$.

Definition A.13 (Inverse sequence)**.** Let $s \in \mathbb{N}^\infty$ be an integer sequence. The inverse sequence $s^{-1} \in \mathbb{N}^\infty$ is the sequence such that $\forall j \leq \|s\| : \oplus s(s^{-1}(j) - 1) < j \leq \oplus s(s^{-1}(j))$. For example, if $s = 1\ 2\ 4$, then $s^{-1}(5) = 3$ because $\oplus s(2) < 5 \leq \oplus s(3)$.

Definition A.14 ($\otimes$ operator)**.** Let $s \in \mathbb{R}^\omega$ and $r \in \mathbb{N}^\omega$ be two infinite sequences. The sequence $s \otimes r \in \mathbb{R}^\omega$ is defined as follows. $\forall i \in \mathbb{N}_{>0} : (s \otimes r)(i) = \sum_{j=1}^{r(i)} s(j + \oplus r(i-1))$.

The `on` operator of the synchronous language `Lucy-n` [134] is equivalent to the $\otimes$ operator defined on binary sequences $\mathbb{B}^\omega$; i.e. the set of infinite sequences over $\mathbb{B} = \{0, 1\}$. The set of ultimately periodic binary sequences corresponds to the set of 2-adic numbers [192]. Table A.1 illustrates the $\otimes$ operator.

Table A.1: Illustration of $\otimes$ operator.

i	1	2	3	4	5	6	...
s	2	3	1	8	6	2	...
$\oplus s$	2	5	6	14	20	22	...
r	0	2	1	0	0	2	...
$\oplus r$	0	2	3	3	3	5	...
$s \otimes r$	0	5	1	0	0	14	...
$\oplus(s \otimes r)$	0	5	6	6	6	20	...

Property A.1. $\oplus(s \otimes r) = (\oplus s) \circ (\oplus r)$.

Proof: From Definition A.14, we note that $\forall i : (s \otimes r)(i) = \oplus s(\oplus r(i)) - \oplus s(\oplus r(i-1))$. Hence, $\oplus(s \otimes r)(i) = \sum_{j=1}^{i} \oplus s(\oplus r(j)) - \sum_{j=1}^{i} \oplus s(\oplus r(j-1)) = \oplus s(\oplus r(i))$. □

Property A.2. The set of ultimately periodic integer sequences $\mathcal{N}$ is closed under the $\otimes$ operation.

Proof: Let $s = u_1 v_1^\omega$ and $r = u_2 v_2^\omega$ be two ultimately periodic sequences. We have that $\forall i > |u_1| : \oplus s(i + |v_1|) = \oplus s(i) + \|v_1\|$ and $\forall i > |u_2| : \oplus r(i + |v_2|) = \oplus r(i) + \|v_2\|$. Let us put

$$\beta = \frac{|v_1|}{\gcd(|v_1|, \|v_2\|)} \quad , \ \alpha = \beta |v_2| \ \ , \ \text{and} \quad \gamma = \frac{\|v_2\|}{\gcd(|v_1|, \|v_2\|)}$$

So, $\forall i > |u_2| : \oplus r(i + \alpha) = \oplus r(i) + \beta \|v_2\| = \oplus r(i) + |v_1| \gamma$. Thus, $\exists i_0 : \forall i \geq i_0$, we have that

$$
\begin{aligned}
(s \otimes r)(i+\alpha) &= \oplus s(\oplus r(i+\alpha)) - \oplus s(\oplus r(i-1+\alpha)) \\
&= \oplus s(\oplus r(i) + \gamma|v_1|) - \oplus s(\oplus r(i-1) + \gamma|v_1|) \\
&= \oplus s(\oplus r(i)) + \gamma\|v_1\| - \oplus s(\oplus r(i-1)) - \gamma\|v_1\| \\
&= (s \otimes r)(i)
\end{aligned}
$$

Therefore, $u_1 v_1^\omega \otimes u_2 v_2^\omega = u v^\omega$ with $|v| = \alpha$ and $\|v\| = \gamma\|v_1\|$. □

Property A.3. $(\mathcal{N}, \otimes, 1^\omega)$ is a monoid.

Proof: 1) Closure. $\forall s, r \in \mathcal{N} : s \otimes r \in \mathcal{N}$ (Property A.2).
2) Identity element. $\forall s \in \mathcal{N} : s \otimes 1^\omega = 1^\omega \otimes s = s$.
3) Associativity. $\forall s, r, t \in \mathcal{N}$:

$$
\begin{aligned}
(s \otimes (r \otimes t))(i) &= \oplus s((\oplus(r \otimes t))(i)) - \oplus s((\oplus(r \otimes t))(i-1)) \\
&= \oplus s(\oplus r(\oplus t(i))) - \oplus s(\oplus r(\oplus t(i-1))) \qquad (\text{ Property A.1}) \\
&= (\oplus(s \otimes r))(\oplus t(i)) - (\oplus(s \otimes r))(\oplus t(i-1)) \\
&= ((s \otimes r) \otimes t)(i)
\end{aligned}
$$

□

List of Figures

1 Real-time scheduling of dataflow graphs. 4
2 (a) un graphe flot de données et (b) une relation d'activation. 8
3 Un ordonnancent abstrait inconsistant. 9
4 Une relation d'activation affine de paramètres $(4, 2, 3)$. 9

1.1 Example of (a) a cyclic SDF graph, and (b) its equivalent HSDF graph. 26
1.2 Reachability graph of the SDF example. 27
1.3 Self-timed execution of the SDF example. 29
1.4 Retiming of a SDF graph. 32
1.5 Pareto space of the SDF example. 35

2.1 Time abstraction of the self-timed example. 54
2.2 Illustration of Proposition 2.1. 57
2.3 Illustration of Lemma 2.2. 58
2.4 Illustration of Proposition 2.2. 59
2.5 Illustration of Proposition 2.3. 60
2.6 Graph of activation relations. 61
2.7 Consistency of symmetric difference of simple cycles. 62
2.8 Illustration of Proposition 2.6. 63
2.9 A $(3, -4, 5)-$affine relation. 68
2.10 A consistent affine schedule. 71
2.11 Function $\mathsf{cbef}_{i,k}$: partitioned fixed-priority scheduling. 73
2.12 Function $\mathsf{cbef}_{k,i}$: partitioned fixed-priority scheduling. 75
2.13 Function $\mathsf{cbef}_{i,k}$: partitioned EDF scheduling, $\nu_i = \nu_k$. 77
2.14 An ultimately cyclo-static dataflow graph. 79
2.15 (a) An UCSDF graph and (b) its equivalent CSDF graph. 79
2.16 Linear bounds of ultimately periodic rates. 80
2.17 Activation relations of the schedule $p_2\ p_3\ p_3\ p_1\ p_2(p_2\ p_3\ p_3\ p_1\ p_3\ p_1)^\omega$. . 81
2.18 (a) An UCSDF graph with a multichannel and (b) its equivalent graph with simple channels. 82
2.19 Buffer minimization using a shared storage space. 83
2.20 A synchronous execution of process p. 85

3.1 Priority assignment problem expressed as a LOP. 92
3.2 Partitioning of the $\mathcal{T}$-space into a set of DM priority regions. 93

3.3 Transforming DM priority assignments. 94
3.4 Exploration of the priority orderings space: throughput vs buffering requirements. 95
3.5 Exploration of the priority orderings space: throughput vs precedence distance. 96
3.6 Exploration of the priority orderings space: throughput vs utilization distance. 96
3.7 Illustration of DF-B&B symbolic FP schedulability analysis. 100
3.8 Totally ordered communication strategy. 103
3.9 $\mathsf{safr}_{i,k}$ function. 103
3.10 Deadlines computation example. 104
3.11 Illustration of SQPA. 107
3.12 Illustration of DF-B&B SQPA. 109
3.13 Illustration of *ffdbf* 111

4.1 Achieved throughput in EDF/RM uniprocessor scheduling. 115
4.2 Impact of number of processors on the throughput in `BF-RM` scheduling. 117
4.3 Comparison between `BF-EDF` and the first fit allocation strategy of DARTS. 117
4.4 Comparison between `BF-EDF` and `FFDBF-SQPA`. 118
4.5 Comparison between DARTS and ADFG in terms of buffering requirements. 119
4.6 Impact of the totally ordered communication strategy on the buffering requirements. 119
4.7 Impact of the LOP priority assignment on the buffering requirements and the processor utilization. 120
4.8 Impact of the constrained LOP priority assignment on the buffering requirements and the processor utilization. 120
4.9 Impact of the priority assignment with utilization distance on the buffering requirements and the processor utilization. 121
4.10 Performance of the `SQPA` algorithm. 122
4.11 Impact of constrained deadlines on the performance of the `SQPA` algorithm. 123
4.12 Performance of the `FFDBF-SQPA` algorithm. 124
4.13 Impact of the number of processors on the performance of the `FFDBF-SQPA` algorithm. 124
4.14 Performance of the `DF-B&B SQPA` algorithm. 125
4.15 Performance of the `SRTA` algorithm. 126
4.16 Performance of the `DF-B&B SRTA` algorithm. 126
4.17 SCJ mission life cycle. 128
4.18 An ultimately cyclo-static dataflow graph with aperiodic actors. 129
4.19 Graphical editor for SCJ/L1 applications design. 130

Abstract

The ever-increasing functional and nonfunctional requirements in real-time safety-critical embedded systems call for new design flows that solve the specification, validation, and synthesis problems. Ensuring key properties, such as functional determinism and temporal predictability, has been the main objective of many embedded system design models. Dataflow models of computation (such as KPN, SDF, CSDF, etc.) are widely used to model stream-based embedded systems due to their inherent functional determinism. Since the introduction of the (C)SDF model, a considerable effort has been made to solve the static-periodic scheduling problem. Ensuring boundedness and liveness is the essence of the proposed algorithms in addition to optimizing some nonfunctional performance metrics (e.g. buffer minimization, throughput maximization, etc.). However, nowadays real-time embedded systems are so complex that real-time operating systems are used to manage hardware resources and host real-time tasks. Most of real-time operating systems rely on priority-driven scheduling algorithms (e.g. RM, EDF, etc.) instead of static schedules which are inflexible and difficult to maintain. This thesis addresses the real-time scheduling problem of dataflow graph specifications; i.e. transformation of the dataflow specification to a set of independent real-time tasks w.r.t. a given priority-driven scheduling policy such that the following properties are satisfied: (1) channels are bounded and overflow/underflow-free; (2) the task set is schedulable on a given uniprocessor (or multiprocessor) architecture. This problem requires the synthesis of scheduling parameters (e.g. periods, priorities, processor allocation, etc.) and channel capacities. Furthermore, the thesis considers two performance optimization problems: buffer minimization and throughput maximization.

Résumé

Les systèmes temps-réel critiques sont de plus en plus complexes, et les exigences fonctionnelles et non-fonctionnelles ne cessent plus de croître. Le flot de conception de tels systèmes doit assurer, parmi d'autres propriétés, le déterminisme fonctionnel et la prévisibilité temporelle. Le déterminisme fonctionnel est inhérent aux modèles de calcul flot de données (ex. KPN, SDF, etc.) ; c'est pour cela qu'ils sont largement utilisés pour modéliser les systèmes embarqués de traitement de flux. Un effort considérable a été accompli pour résoudre le problème d'ordonnancement statique périodique et à mémoire de communication bornée des graphes flots de données. Cependant, les systèmes embarqués temps-réel optent de plus en plus pour l'utilisation de systèmes d'exploitation temps-réel et de stratégies d'ordonnancement dynamique pour gérer les tâches et les ressources critiques. Cette thèse aborde le problème d'ordonnancement temps-réel dynamique des graphes flot de données ; ce problème consiste à assigner chaque acteur dans un graphe à une tâche temps-réel périodique (i.e. calcul des périodes, des phases, etc.) de façon à : (1) assurer l'ordonnançabilité des tâches sur une architecture et pour une stratégie d'ordonnancement (ex. RM, EDF) données ; (2) exclure statiquement les exceptions d'overflow et d'underflow sur les buffers de communication ; et (3) optimiser les performances du système (ex. maximisation du débit, minimisation des tailles des buffers).