Ordonnancement temps réel hors-ligne optimal à l'aide de réseaux de Petri en environnement monoprocesseur et multiprocesseur

Emmanuel Grolleau

▶ To cite this version:

Emmanuel Grolleau. Ordonnancement temps réel hors-ligne optimal à l'aide de réseaux de Petri en environnement monoprocesseur et multiprocesseur. Informatique [cs]. Université de Poitiers, 1999. Français. NNT : . tel-04357588

HAL Id: tel-04357588

https://theses.hal.science/tel-04357588

Submitted on 21 Dec 2023

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.

Distributed under a Creative Commons Attribution - NoDerivatives 4.0 International License

THESE

Pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITE DE POITIERS

Ecole Nationale Supérieure de Mécanique et d'Aérotechnique

Faculté des Sciences Fondamentales et Appliquées

Ecole Doctorale des Sciences pour l'Ingénieur

(Diplôme National – Arrêté du 30 Mars 1992)

Secteur de Recherche : INFORMATIQUE

Présentée et soutenue publiquement par

Emmanuel GROLLEAU

Le 29 Novembre 1999

Ordonnancement temps réel hors-ligne optimal à l'aide de réseaux de Petri en environnement monoprocesseur et multiprocesseur.

Directeurs de Thèse : Francis COTTET, Annie CHOQUET-GENIET

JURY

Président :	P. Estraillier	Professeur, Université de La Rochelle, L3I
Rapporteurs :	G. Juanole	Professeur, Université Paul Sabatier, LAAS
	G. Vidal-Naquet	Professeur, Supélec, LRI
Examinateurs :	A. Choquet-Geniet	Maître de Conférences, Université de Poitiers, LISI
	F. Cottet	Professeur, ENSMA, LISI
	A-M. Deplanche	Maître de Conférences, Université de Nantes, IRCyN

LABORATOIRE D'INFORMATIQUE SCIENTIFIQUE ET INDUSTRIELLE
Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2 - 1, avenue Clément Ader - BP 40109 - 86961 Futuroscope
Tél: 05.49.49.80.63 - Fax: 05.49.49.80.64

Remerciements

J'exprime mes plus vifs remerciements à Annie Choquet-Geniet pour avoir encadré mon travail de thèse. Ses conseils constants m'ont été une aide inestimable et ont largement contribué à ma formation de chercheur.

Je tiens à remercier le Professeur Francis Cottet, qui m'a permis d'intégrer l'équipe Temps Réel, pour les nombreuses et fructueuses discussions sur mon travail.

Cette thèse est le fruit d'un travail mené au sein du Laboratoire d'Informatique Scientifique et Industrielle dirigé par le Professeur Guy Pierra, à qui je tiens à exprimer toute ma gratitude pour les moyens techniques et scientifiques qu'il a mis à ma disposition.

Je tiens à remercier le Professeur Guy Juanole et le Professeur Guy Vidal-Naquet qui ont accepté la lourde charge de rapporteurs, ainsi que Anne-Marie Deplanche et le Professeur Pascal Estraillier pour l'honneur qu'ils m'ont fait en acceptant d'être examinateurs de mon jury.

J'adresse un remerciement particulier à Dominique Geniet, qui, grâce à sa relecture très attentive de mon rapport, m'a permis d'en améliorer sensiblement la qualité, et à Yamine Aït-Ameur, Maximilien Holle ainsi qu'à Pascal Richard pour les discussions fructueuses sur le plan scientifique, que nous avons eues.

Je remercie tous les membres du laboratoire, et particulièrement Manu, Ricou, Dag, Tex, Fabricio, JCP, Laurent et Dave pour les bon moments que nous avons passés ensemble, ce qui m'a permis d'affronter la route et le rail quotidiennement.

Enfin, un grand merci à Patricia qui m'a soutenu et encouragé pendant toute la durée de ce travail.

SOMMAIRE

Introduction générale 2

ETAT DE L'ART 3

I Systèmes temps réel et ordonnancement 4

I-1 Introduction au temps réel 5

I-1.1 Le contrôle de procédé 6
I-1.2 Hypothèse synchrone et asynchrone 6
I-1.2.1 Hypothèse synchrone 6
I-1.2.2 L'hypothèse asynchrone 7
I-1.3 Les systèmes temps réel 9
I-1.3.1 L'exécutif temps réel 9
I-1.3.1.a Gestion des tâches 9
I-1.3.1.b Gestion des communications et des synchronisations 10
I-1.3.1.c Gestion des ressources critiques 14
I-1.3.1.d La gestion du temps 16
I-1.3.2 Contextes temps réel 17
I-1.3.3 Tâches temps réel 17
I-1.3.3.a Tâches périodiques 17
I-1.3.3.b Tâches non périodiques 21
I-1.3.3.c Contextes d'ordonnancement 23
I-1.4 Conclusion 25

I-2 L'ordonnancement des systèmes de tâches temps réel 27

I-2.1 Propriétés générales des algorithmes d'ordonnancement 27
I-2.1.1 Notions de validité, de puissance, et d'optimalité 27
I-2.1.2 Validation temporelle de systèmes temps réel 29
I-2.1.2.a Approches en-ligne 30
I-2.1.2.b Approches hors-ligne 30
I-2.1.2.c Validation par simulation 31
I-2.2 Ordonnancement en-ligne 32
I-2.2.1 Ordonnancement en-ligne de tâches indépendantes 32
I-2.2.1.a Algorithmes à priorités statiques 32
I-2.2.1.b Algorithmes à priorités variables 38
I-2.2.1.c Récapitulatif sur l'ordonnancement de tâches indépendantes en environnement monoprocesseur 42
I-2.2.2 Ordonnancement en-ligne de tâches communicantes 42
I-2.2.3 Partage de ressources 46
I-2.2.3.a Le protocole à priorité héritée (PPH) 49
I-2.2.3.b Le protocole à priorité plafond (PPP) 51
I-2.2.3.c Le protocole d'allocation de la pile (PAP) 54
I-2.2.3.d Conclusion sur la gestion de ressources dans les algorithmes en-ligne 55
I-2.3 Ordonnancement et complexité en environnement multiprocesseur 56
I-2.4 Approches hors-ligne 59
I-2.5 Conclusion 63

II Extensions des Réseaux de Petri 64

II-1 Réseaux de Petri autonomes 65

II-1.1 Définitions et notations 65
II-1.2 Langage d'un réseau de Petri 67
II-1.3 Réseaux de Petri colorés 69

II-2 Réseaux de Petri prenant en compte le temps 71

II-2.1 Réseaux de Petri temporels ____ 71
II-2.1.1 Définitions et notations ____ 71
II-2.1.2 Puissance d'expression ____ 73
II-2.2 Réseaux de Petri temporisés ____ 75
II-2.3 Réseaux de Petri autonomes avec la règle de tir maximal ____ 78
II-2.4 Conclusion ____ 79

CONTRIBUTION ____ 81

III Etude de la Cyclicité des Ordonnancements de Tâches Périodiques ____ 82

III-1 Etude des temps creux acycliques ____ 85

III-1.1 Cas des systèmes de tâches indépendantes ____ 87
III-1.1.1 La date d'occurrence du dernier temps creux acyclique est bornée ____ 90
III-1.1.2 L'état d'un système est identique après le dernier temps creux acyclique et une méta-période plus tard ____ 91
III-1.2 Cas des systèmes de tâches quelconques ____ 91
III-1.2.1 Définition des chaînes d'attente ____ 92
III-1.2.2 Périodicité des chaînes d'attente ____ 95
III-1.2.3 Cyclicité des ordonnancements dans le cas général ____ 96
III-1.2.3.a La date d'occurrence du dernier temps creux acyclique est bornée ____ 97
III-1.2.3.b L'état d'un système est identique après le dernier temps creux acyclique et une méta-période plus tard ____ 98

III-2 Conclusion ____ 103

III-2.1 Cas des algorithmes conservatifs ____ 103
III-2.2 Cas des algorithmes non conservatifs ____ 106

IV Etude de Systèmes Temps Réel à l'Aide de Réseaux de Petri: Cas Monoprocesseur 110

IV-1 Modélisation ____ 111

IV-1.1 Hypothèses générales ____ 111
IV-1.2 Principe général ____ 113
IV-1.3 Système d'horlogerie ____ 114
IV-1.4 Systèmes de tâches ____ 116
IV-1.4.1 Modélisation des blocs indépendants ____ 116
IV-1.4.2 Modélisation des communications ____ 118
IV-1.4.3 Modélisation des ressources critiques ____ 119
IV-1.4.4 Schéma global d'une tâche ____ 120
IV-1.5 Prise en compte des temps creux ____ 122
IV-1.6 Prise en compte des dates de réveil ____ 122
IV-1.7 Prise en compte des contraintes temporelles ____ 123
IV-1.7.1 Tâches à échéance avant requête ____ 123
IV-1.7.2 Tâches à échéance sur requête ____ 124
IV-1.7.3 Cas général ____ 124
IV-1.8 Récapitulatif ____ 125
IV-1.9 Exemples de modélisation ____ 127
IV-1.9.1 Tâches à échéance sur requête à départ simultané, avec une ressource en écriture et une boîte aux lettres ____ 127
IV-1.9.2 Illustration de l'obtention des séquences non conservatives ____ 128
IV-1.9.3 Tâches comportant des parties non préemptibles ____ 129

IV-2 Etude ____ 133

IV-2.1 Profondeur du graphe d'accessibilité ____ 133
IV-2.2 Représentation du graphe d'accessibilité ____ 134

IV-2.2.1 Deux tâches simultanées indépendantes de charge 100% 134
IV-2.2.2 Deux tâches différées indépendantes de charge 100% 134
IV-2.3 Complexité et taille du graphe d'accessibilité 135
IV-2.3.1 Borne théorique à la taille du graphe d'accessibilité 136
IV-2.3.2 Structure de données utilisée pour stocker et rechercher les marquages 139
IV-2.4 Obtention des séquences d'ordonnancement valides 141
IV-2.4.1 Construction du graphe des marquages 142
IV-2.4.1.a Construction en largeur 142
IV-2.4.1.b Construction en profondeur 143
IV-2.4.1.c Comparatif d'une construction en largeur par rapport à en profondeur 143
IV-2.4.2 Diminution des cas de retours arrière 144
IV-2.4.3 Diminution de la taille du graphe des marquages 146
IV-2.4.3.a Contraintes de successeur 146
IV-2.4.3.b Contraintes absolues 147
IV-2.4.4 Extraction de séquences d'ordonnancement optimales 149
IV-2.4.4.a Exécution au plus tôt 150
IV-2.4.4.b Optimisation du temps de réponse 151
IV-2.4.4.c Optimisation du taux de réaction 153
IV-2.4.4.d Optimisation du retard 154
IV-2.4.4.e Minimisation de la gigue 155
IV-2.4.4.f Répartition des temps creux 160
IV-2.4.4.g Discussion sur les critères qualitatifs 161

V Etude de Systèmes Temps Réel à l'Aide de Réseaux de Petri: Cas Multiprocesseur 164

V-1 Hypothèses matérielles 165

V-2 Modélisation 167

V-3 Etude 171

V-3.1 Profondeur du graphe d'accessibilité 171
V-3.2 Représentation du graphe d'accessibilité 171
V-3.2.1 Deux tâches simultanées indépendantes de charge 100% 171
V-3.2.2 Deux tâches simultanées indépendantes de charge inférieure à 100% 172
V-3.3 Taille du graphe d'accessibilité 173
V-3.4 Obtention des séquences d'ordonnancement valides 174
V-3.4.1 Placement des tâches a posteriori 174
V-3.4.2 Diminution des cas de retours arrière 174
V-3.4.3 Note sur les contraintes de successeur 175
V-3.4.4 Diminution de la taille du graphe des marquages 176
V-3.4.5 Extraction de séquences d'ordonnancement optimales 176

VI Application de l'Etude 178

VI-1 Présentation de *PeNSMARTS* 179

VI-1.1 Saisie du système de tâches à étudier 179
VI-1.2 Génération du modèle RdP 181
VI-1.3 Génération des séquences d'ordonnacement 182

VI-2 Etude de cas 185

VI-2.1 Description du procédé 185
VI-2.2 Spécification du contrôle 186
VI-2.3 Implémentation 187

Conclusion 192

Bibliographie ______ **196**

Bibliographie liée à l'étude ______ **206**

Annexes ______ **208**

I-1 Etude du *ppcm* ______ 209

I-2 Notions de complexité ______ 211

I-3 Fonctionnement d'un réseau de Petri avec la règle de tir maximal ______ 215

I-4 Preuves relatives au résultat de cyclicité des ordonnancements ______ 219

I-5 Algorithme de placement de tâches évitant les changements de contexte inutiles ______ 225

Index ______ **226**

Index des Figures ______ **230**

INTRODUCTION GÉNÉRALE

Les systèmes temps réels sont de plus en plus présents dans le quotidien. Systèmes informatiques reliés à un procédé à l'aide de capteurs et d'actionneurs, leur évolution doit se calquer sur leur environnement. Leur essence est donc leur réactivité : tout doit se passer comme s'ils réagissaient en temps réel par rapport au procédé contrôlé. Cela implique en premier lieu la nécessité pour le concepteur de faire en sorte que le programme respecte des contraintes temporelles inhérentes à l'environnement.

Différentes études ont été menées au cours de ces trente dernières années, certaines se basant sur l'hypothèse synchrone (le temps de traitement informatique est très court par rapport à l'évolution du procédé), et les autres sur l'hypothèse asynchrone.

Les applications temps réel, sous l'hypothèse asynchrone, sont subdivisées en tâches informatiques de durée non nulle. La problématique principale, dans ce cas, est de s'assurer de la correction temporelle des tâches présentes. Deux grands courants ont émergé en ce qui concerne la validation temporelle : l'approche en-ligne et l'approche hors-ligne. La première approche se base sur l'utilisation d'algorithmes de complexité polynomiale, qui sont optimaux dans certains cas, mais seulement approchés dans le cas général. La seconde approche, que nous développons dans ce document, se base sur une étude des tâches généralement plus coûteuse en temps, permettant dans tous les cas de trouver comment agencer les tâches, on parle alors d'ordonnancement, afin d'assurer le respect des contraintes temporelles.

L'avantage de la première approche est la simplicité de mise en œuvre des algorithmes, mais en contre-partie, dès lors que les tâches interagissent, les applications sont difficiles, voire impossibles à valider.

Cet inconvénient est absent de certaines approches hors-ligne, qui peuvent être optimales. Cependant, le coût de l'optimalité se traduit par la difficulté relative de mise en œuvre de telles approches, et surtout par un temps de calcul au moins exponentiel. Ce désagrément entraîne la mise en œuvre d'optimisations et d'heuristiques diverses, choisies pour nuire le moins possible à l'ordonnançabilité des systèmes de tâches étudiés.

Etant donné que le coût des approches optimales est au moins exponentiel, nous nous sommes intéressés à la recherche d'ordonnancements ayant certaines caractéristiques qualitatives et quantitatives, qui peuvent être obtenues au même coût.

Notre approche se base sur une énumération des séquences d'ordonnancement valides d'un système de tâches. Afin de favoriser l'évolution des systèmes considérés par notre approche, nous avons choisi de baser cette énumération sur un réseau de Petri, construit de sorte que son langage soit l'ensemble des ordonnancements valides du système modélisé. Ainsi, il a été possible de prendre en compte dans le modèle différents modes d'interactions entre tâches qui, à notre connaissance, n'ont jamais été tous pris en compte dans la même approche. Une grande partie de notre étude a consisté à trouver des optimisations et heuristiques permettant d'obtenir des ordonnancements valides en diminuant le temps et l'espace nécessaires à leur obtention. Enfin, au cours de notre étude est apparu un problème ouvert concernant la durée nécessaire à l'étude d'un ordonnancement lorsque les tâches sont initialement différées. L'étude réalisée grâce aux graphes des marquages générés par notre modèle nous a permis de résoudre cette question.

Dans le premier chapitre, nous exposons la problématique temps réel, et présentons les hypothèse synchrones et asynchrones, ainsi que les noyaux temps réel servant de support de développement aux applications temps réel. Ensuite, nous exposons certaines approches proposées dans la littérature, en en distinguant les deux grands courants : en-ligne et hors-ligne.

Le second chapitre présente quelques extensions des réseaux de Petri, afin que les lecteurs non familiers avec ce modèle puissent appréhender la suite.

Le troisième chapitre résout le problème, ouvert jusqu'ici, de la durée de simulation nécessaire à l'étude de systèmes de tâches dont certaines peuvent être initialement différées.

Le quatrième chapitre présente le modèle réseau de Petri utilisé pour énumérer les ordonnancements valides des systèmes de tâches dans le cas monoprocesseur. Son langage est étudié à travers l'allure de son graphe des marquages. Sa construction, et l'utilisation d'heuristiques permettant de diminuer sa taille sans nuire à l'ordonnançabilité du système étudié sont présentées. Nous expliquons de quelle manière il est possible d'extraire simplement des ordonnancements ayant des propriétés qualitatives et quantitatives grâce à la forme particulière de ce graphe.

Le cinquième chapitre décrit les modifications mineures qui ont été proposées afin d'adapter le modèle réseau de Petri utilisé aux environnements multiprocesseur.

Enfin, le sixième chapitre détaille une étude de cas traitée à l'aide de la méthodologie proposée.

PARTIE I

ETAT DE L'ART

I Systèmes temps réel et ordonnancement

Cette partie introduit la problématique temps réel à travers la présentation des hypothèses synchrone et asynchrone. Puis elle aborde le problème sous l'hypothèse asynchrone par la description d'un noyau temps réel type : support de développement d'applications temps réel, sa présentation permet d'appréhender les éléments devant être pris en compte dans une étude temporelle, ce qui permet d'introduire les différents modèles de tâches temps réel.

I-1 Introduction au temps réel

L'émergence de l'informatique temps réel va de pair avec l'informatisation du contrôle de procédés. Les domaines concernés vont des chaînes de production aux transports aériens en passant par les robots de plus en plus perfectionnés, tels que les modules d'exploration planétaire, et l'aide à la conduite automobile. Les débouchés sont considérables et à l'aube du vingt-et-unième siècle, nombreux sont les projets d'automatisation partielle ou complète de nombreuses tâches, telles que la conduite automobile ou le contrôle de chaînes de montage.

Le qualificatif **temps réel** désigne toute application mettant en œuvre un système informatique dont le fonctionnement est assujetti à l'évolution dynamique de l'état d'un environnement (appelé procédé) qui lui est connecté et dont il doit contrôler le comportement [CNRS 88]. Il peut qualifier le mouvement d'une caméra pilotée à distance qui commencera quelques centaines de millisecondes après une commande du télé opérateur, ou bien le processus de décision de stabilisation d'un avion qui ne devra pas dépasser une cinquantaine de millisecondes. Il désigne une réaction dont le temps de réponse est relativement court par rapport à la vitesse d'évolution du procédé [TE 97].

Pour certaines classes de systèmes informatisés (comme les systèmes interactifs ou transformationnels), le temps de réponse des programmes n'influe que sur le confort d'utilisation, et, si un utilisateur manque de patience, il peut investir dans du matériel informatique plus puissant ou un réseau à haut débit.

Les systèmes temps réel, quant à eux, doivent maintenir une cohérence la plus fine possible d'une part entre leur environnement et la représentation interne qu'ils en ont, et d'autre part entre une modification d'état de leur représentation interne du monde et leur réaction face à ce changement. Cette contrainte est d'une part logicielle (le système doit être correctement programmé), mais aussi temporelle : il est primordial que le système informatique temps réel puisse réagir dans un temps relativement court (on peut parler alors de réaction instantanée pour un observateur extérieur) par rapport à la vitesse d'évolution de son environnement. En effet, un robot manipulateur de chaîne de montage doit réagir en un temps relativement court par rapport à la vitesse de passage de la pièce détachée devant ses capteurs, car son bras mécanique doit saisir une pièce en mouvement.

La notion d'instantanéité de la réaction d'un système temps réel est une vue idéale, elle se traduit dans la réalité par la nécessité de respecter des contraintes temporelles induites par la dynamique de l'environnement du procédé contrôlé.

La différence marquante entre les différents contextes que l'on pourrait qualifier de temps réel est la notion de criticité du temps. En effet, alors que le retard de quelques millisecondes d'une trame vidéo sur un réseau ne fera que nuire au confort d'un utilisateur, le retard de la décision de redresser un avion peut provoquer une catastrophe sur le plan humain ainsi que des pertes financières considérables [Sta 88]. On parle alors de temps réel mou, comme c'est le cas pour les applications multimédias, et de temps réel dur (ou strict), comme c'est le cas pour les systèmes critiques. Dans cet ouvrage, nous nous intéresserons au temps réel dur.

I-1.1 Le contrôle de procédé

Les **systèmes temps réel** sont généralement composés de deux éléments distincts : une ou plusieurs entités physiques constituent le **procédé**, dont le rôle est d'agir et de « sentir », et un contrôle informatique, nommé **contrôleur** ou **application temps réel** qui est le cerveau du procédé. Le contrôleur reçoit des informations sur l'environnement du procédé à l'aide de **capteurs** et commande les changements d'état du procédé via des **actionneurs**.

La figure I-1-1 donne un aperçu des interactions qui existent entre procédé et contrôleur d'un système temps réel.

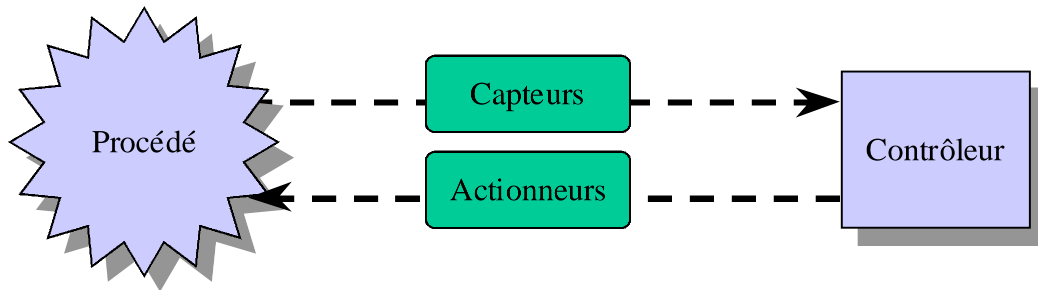


figure I-1-1 : interactions entre procédé et contrôleur

Les informations collectées par les capteurs peuvent être utilisées par le contrôleur soit sous la forme d'événements, soit sous la forme de données continues, qui peuvent être lues explicitement par le contrôleur. On retrouve les événements discrets et continus de *SA-RT* [Gol 93].

Lorsqu'il est confronté à l'implémentation d'un système temps réel dédié au contrôle d'un procédé, un ingénieur a le choix entre l'approche synchrone et l'approche asynchrone.

Dans certains cas, la vitesse de traitement de l'information est telle comparée aux modifications de l'environnement qu'il est possible de faire l'hypothèse de l'instantanéité entre action et réaction : c'est l'**hypothèse synchrone**, faite dans les langages réactifs. Dans d'autres cas, la masse d'informations à traiter est telle que cette hypothèse n'est plus faisable : c'est l'**hypothèse asynchrone**.

I-1.2 Hypothèse synchrone et asynchrone

I-1.2.1 Hypothèse synchrone

L'hypothèse synchrone suppose que le temps des calculs est négligeable par rapport à la dynamique du procédé. Avec une telle hypothèse, il est inutile d'interrompre un traitement par un autre jugé plus prioritaire, puisqu'il est tellement rapide qu'il peut être considéré de durée nulle [Ell 97].

La figure I-1-2 montre le principe de l'approche synchrone : les événements ne sont observés par le système de tâches que lorsqu'aucun événement précédent n'est en cours de traitement, c'est à dire uniquement lorsqu'ils peuvent être traités. Si un événement a lieu lors d'un traitement, une implémentation possible est de considérer le système comme éronné puisqu'il ne respecte pas l'hypothèse synchrone, ou bien de ne percevoir l'événement qu'à la fin du traitement en cours. Tout se passe donc comme si le système réagissait de façon synchrone avec les événements extérieurs.

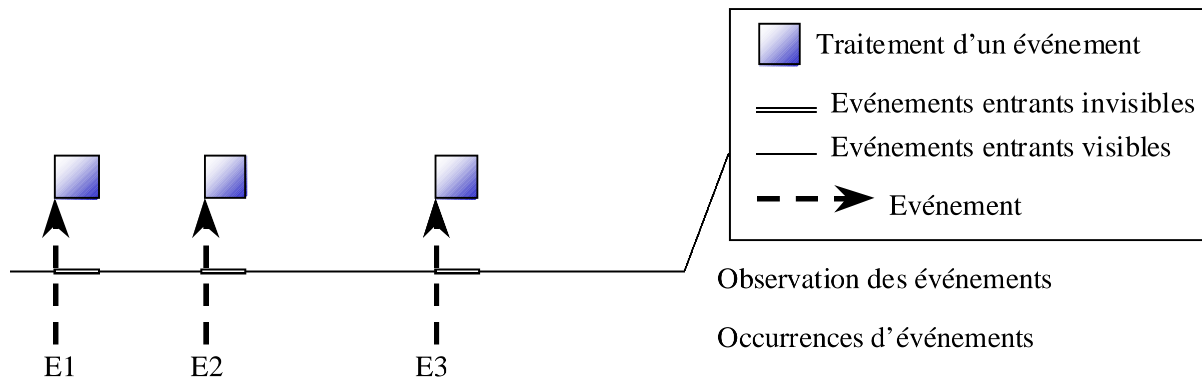


figure I-1-2 : chronogramme du comportement d'un système temps réel synchrone

L'approche synchrone est exemptée des problèmes liés aux exclusions mutuelles entre les tâches. De plus, les durées de traitement des événements extérieurs étant considérées comme nulles par rapport à la fréquence d'arrivée des événements, le problème de l'ordonnancement des tâches (i.e. ordre dans lequel elles s'exécutent) n'est pas important.

Un autre avantage de l'approche synchrone est sa simplicité relative de programmation dans l'utilisation des langages synchrones. Dans les **langages synchrones** (appelés aussi **langages réactifs**) on peut citer ESTEREL [BG 92][Ber 98a][Ber 98b], LUSTRE [HCRP 91][HLR 92] et SIGNAL [GBGM 91] qui sont tous basés sur une hypothèse de temps de traitement informatique nul au niveau du contrôleur. Le temps n'est pas quantitatif mais relatif aux événements, on dit qu'il est guidé par les événements. Des propriétés, notamment de vivacité et de sûreté, sur les programmes générés peuvent être montrées à l'aide de techniques basées sur le *model checking* [CGL 93][Mill 93]. La complexité moyenne de ce genre d'étude, pouvant être exhaustive, est fortement réduite par l'utilisation de *Binary Decision Diagrams* (ou *BDD*) [Bry 92] qui permettent de diminuer sensiblement la taille des graphes d'états générés en se basant sur une relative indépendance existant entre les différents états du contrôle.

Cependant, bien que l'évolution des processeurs soit rapide, la complexité des procédés à contrôler va croissante. De plus, la fabrication à grande échelle de systèmes temps réel, notamment en ce qui concerne l'industrie automobile, nécessite l'utilisation de processeurs à moindre coût. Lorsque l'hypothèse synchrone ne peut être posée, un concepteur d'applications temps réel doit se tourner vers l'approche asynchrone.

I-1.2.2 L'hypothèse asynchrone

Lors d'une approche asynchrone, aucune hypothèse de puissance de traitement informatique n'est formulée. Chaque action nécessite une durée d'exécution non nulle, et dans ce cas, il est nécessaire de démontrer que le contrôleur ne conduira pas le procédé dans des états indésirables (au crash d'un avion par exemple) par sa lenteur. Il est donc primordial qu'un système temps réel soit prédictible, afin qu'une étude puisse montrer sa correction temporelle.

Dans le cadre d'une approche asynchrone d'un système temps réel, les événements doivent être pris en compte le plus rapidement possible, et peuvent intervenir à n'importe quel moment, y compris lors du traitement d'un événement antérieur (voir figure I-1-3). Il est donc désirable que le programme puisse changer de comportement en interrompant le traitement en cours afin de consacrer les ressources informatiques à un événement jugé important : on parle alors de préemption.

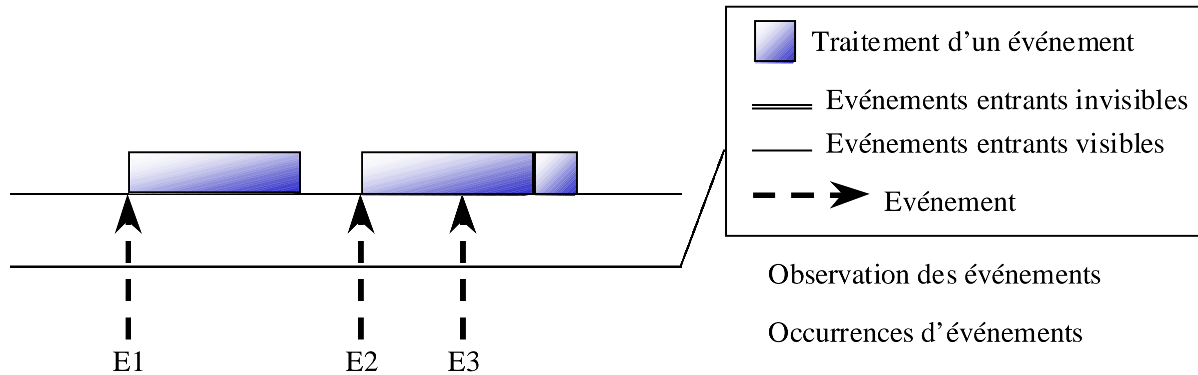


figure I-1-3 : chronogramme du comportement d'un système temps réel asynchrone : les événements entrants sont toujours visibles

Définition I-1-1 : **préemption**

Action d'interrompre une tâche qui utilise le processeur afin d'en exécuter une autre.

Le problème est que dans ce cas, le traitement d'un événement peut arbitrairement retarder celui d'un autre qui, bien que moins important, n'en est pas forcément moins vital au comportement du programme. Deux constats ont été faits à partir de ce problème :

Premièrement, il est souhaitable de pouvoir interrompre une action pour en commencer une autre afin de permettre d'interrompre un traitement par un autre. Cela est fait en séparant les actions du programme temps réel en **tâches informatiques**. Ainsi, une tâche pourra interrompre une autre tâche sans pour autant perdre le travail effectué par la tâche interrompue : c'est la préemption.

Deuxièmement, il est nécessaire d'assurer que, quelle que soit l'évolution du monde dans lequel se situe le procédé, et donc quelles que soient les occurrences d'événements, le programme réagit en temps relativement rapide.

Le temps est dans cette hypothèse un temps physique, donné par une horloge. Les ordinateurs ne pouvant préempter les tâches qu'à intervalles de temps discrets, il s'ensuit une discrétisation du temps : les durées des actions sont exprimées en unités de temps entières ou rationnelles, que l'on peut ramener à des durées entières.

L'expression « en temps relativement rapide » traduit d'une part la nécessité de conserver une cohérence entre le monde extérieur et le modèle interne du programme et d'autre part l'assurance d'une réaction « en temps réel » aux sollicitations extérieures. Cette notion est évidemment relative à la dynamique du procédé : four à verre dont la température et les niveaux doivent être régulés ou bien avion dont l'altitude doit être modulée. La notion de temps réel doit donc être traduite en contraintes temporelles, déduites d'un cahier des charges décrivant le comportement voulu du procédé. Ces contraintes temporelles sont appliquées sur les tâches du programme puisque ce sont ces dernières qui réagissent aux évolutions du procédé. Les tâches sont donc qualifiées de tâches temps réel, et il est indispensable de prouver leur correction temporelle afin de démontrer que le procédé aura toujours un comportement cohérent avec ce qu'exigeait le cahier des charges.

Puisque la correction temporelle des tâches temps réel dépend du retard qu'elles peuvent s'infliger à cause de leur exécution concurrente, l'ordre dans lequel elles s'exécutent est primordial pour la correction temporelle du programme. C'est cet ordre d'exécution, appelé **ordonnancement**, qui est étudié dans ce travail.

Définition I-1-2 : Ordonnancement

Ordre d'exécution des tâches sur un ou plusieurs processeurs.

I-1.3 Les systèmes temps réel

Afin de mieux cerner les problèmes posés par l'étude des systèmes temps réel, la section suivante décrit un noyau temps réel typique.

I-1.3.1 L'exécutif temps réel

Un **exécutif temps réel** est un système d'exploitation multitâche dans lequel la durée d'exécution de toute fonction système est documentée (ou bien au moins prédictible et connue). Il est ainsi possible d'évaluer la durée de chaque tâche à partir de son code source, ce qui permet l'étude temporelle des systèmes implémentés sur un exécutif temps réel.

L'exécutif est chargé de faire le lien entre les applications, qui devraient être indépendantes de la plate-forme support, et l'architecture physique de la plate-forme support. Il est accompagné d'outils de développement tels que des compilateurs *C* ou *Ada*.

Lors d'une approche asynchrone, les langages *C* et *Ada* sont les plus utilisés en informatique temps réel. Ils sont donc utilisés dans la suite comme support de description des **primitives temps réel** fournies par un **noyau temps réel** minimaliste. Ce noyau servira de base aux exemples d'applications temps réel développés au long de cet ouvrage.

Les langages basés sur un exécutif temps réel permettent de gérer des tâches, de les faire communiquer, se synchroniser, partager des ressources, et bien entendu de gérer le temps. Quatre types de primitives, dites temps réel, sont donc proposées dans ces langages :

- La gestion des tâches
- La gestion des communications et synchronisations
- La gestion des ressources critiques
- La gestion du temps

Dans la suite, un sous-ensemble des primitives généralement proposées est présenté. Ce sous-ensemble est volontairement restreint, car une description exhaustive des possibilités des noyaux temps réel n'est pas ici le sujet. Les primitives sont inspirées des primitives trouvées dans le système *RTEMS* [RTE 96] pour le *C* et dans le langage *Ada95* [Ada 95][Bar 96].

I-1.3.1.a Gestion des tâches

Les tâches atteignent au cours de la vie d'une application les états décrits sur la figure I-1-4 [RTE 96].

Une tâche est initialement **créée**, ainsi, elle devient **existante** mais **endormie**. Elle doit être **réveillée**, après initialisation, par son **réveil**, qui la met dans l'état « **prête** » (on dit alors qu'elle est **active**). C'est dans cet état qu'elle requiert un processeur. Lorsque le noyau le décide, suivant la politique d'ordonnancement choisie, cette tâche se verra allouer un processeur afin d'être **exécutée**. De l'état « exécutée », une tâche peut être **préemptée** par une autre, ou bien se **mettre en attente** d'un message, d'une date, d'un événement, ou bien de l'accès à une ressource. Lorsqu'elle est en attente, une tâche passe dans l'état « prête » lorsque l'événement, la date, ou le message (resp. la ressource) est arrivé (resp. libre). Il est à noter qu'une tâche en attente d'une ressource qui n'arrivera jamais est **bloquée**, on parle alors de **blocage fatal** ou de ***deadlock***.

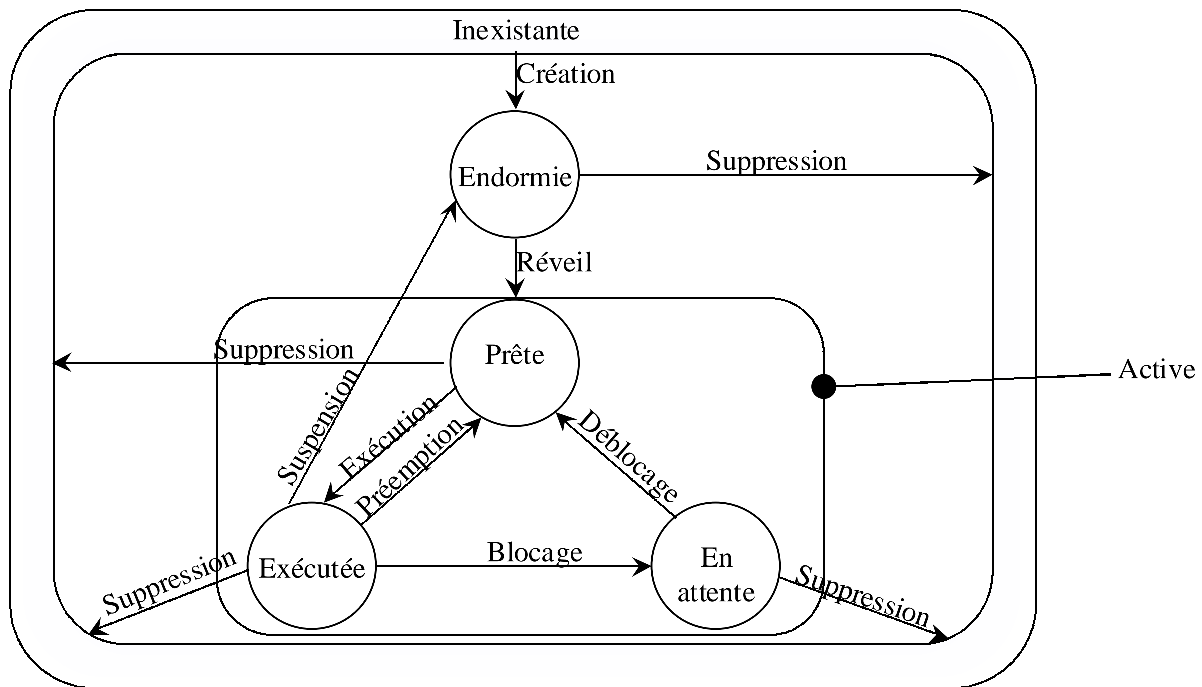


figure I-1-4 : graphe des états possibles des tâches gérées par un noyau temps réel

A tout instant, une tâche peut être **supprimée**. De plus, une tâche peut se mettre en attente pendant un certain temps ou jusqu'à une certaine date, on parle de « **suspension** ».

Afin de gérer les tâches, les noyaux temps réel proposent au moins le jeu d'instruction présenté dans le tableau I-1-1.

Instruction	Paramètres	Post-condition
Creer_Tâche	τ : nom de tâche χ : code source composant la tâche	une tâche τ est créée, elle est dans l'état « endormie »
Lancer_Tâche	τ : nom de tâche	τ passe de l'état « endormie » à l'état « prête », son code est prêt à être exécuté
Supprimer_Tâche	τ : nom de tâche	la tâche τ est supprimée

tableau I-1-1 : primitives de gestion des tâches dans un noyau temps réel minimaliste

Ces primitives sont transparentes en *Ada* : en effet puisque la création d'une tâche a lieu lors de la déclaration d'un objet de type tâche, son lancement a lieu après sa création et son initialisation, et sa suppression a lieu dès que son code est terminé.

Les primitives de gestion de priorités sont volontairement mises de côté, car la gestion des priorités est inhérente à l'ordonnancement choisi. L'ordonnancement est étudié largement dans la suite.

I-1.3.1.b Gestion des communications et des synchronisations

i) Communication asynchrone : La boîte aux lettres

Les **communications asynchrones** se font à travers un tampon d'échange permettant à des tâches d'échanger des données. La zone tampon porte le nom de **boîte aux lettres** puisque

l'analogie existant entre communication asynchrone et communication par boîte postale permet d'appréhender très simplement le concept.

La zone d'échange est une boîte aux lettres dans laquelle une tâche dite **émettrice** peut déposer des données. Les données de la boîte aux lettres sont gérées en *fifo* (i.e. premier déposé/premier retiré), et la tâche dite **réceptrice** retire les données dans l'ordre d'arrivée.

On parle de communication asynchrone car, en règle générale, une tâche émettrice n'a pas besoin d'attendre que la tâche réceptrice soit à l'écoute pour lui envoyer des données.

Cependant, les communications par boîtes aux lettres présentent quelques contraintes :

Une boîte aux lettres a une taille bornée, et lorsqu'une tâche doit déposer des données dans une boîte pleine, suivant l'implémentation choisie, soit le message le plus ancien est écrasé par le nouveau (on peut alors parler de tableau noir), soit la tâche est bloquée jusqu'à ce que des données aient été retirées de la boîte. Si la boîte est vide, une tâche désirant recevoir des données est mise en attente (état « en attente ») jusqu'à ce que des données aient été déposées dans la boîte.

Les primitives de communication asynchrone proposées par les noyaux temps réel sont présentées dans le tableau I-1-2.

Instruction	Paramètres	Post-condition
Créer_BAL	β : nom de boîte aux lettres t : taille type : avec écrasement ou non	une boîte aux lettres β, de taille t est créée avec écrasement des messages ou non lorsque la boîte est pleine.
Déposer_BAL	β : nom de boîte aux lettres δ : donnée	la tâche exécutant cette instruction (émettrice), dépose la donnée δ dans β. Suivant l'implémentation, cette instruction peut être bloquante : si β est pleine, soit l'émettrice passe dans l'état « en attente » jusqu'à ce que δ puisse être déposée dans β, ou non dans le cas où le plus vieux message est écrasé.
Retirer_BAL	β : nom de boîte aux lettres renvoie δ : donnée	la tâche exécutant cette instruction (réceptrice), retire une donnée δ de β. Cette instruction est bloquante : si β est vide, la réceptrice passe dans l'état « en attente » jusqu'à ce que δ puisse être retirée. Généralement, les tâches en attente sur une boîte aux lettres sont mémorisées dans une file *fifo* gérée avec ou sans priorités.

tableau I-1-2 : primitives de communication asynchrone dans un noyau temps réel minimaliste

Les communications asynchrones ne sont pas offertes directement dans le jeu d'instruction standard d'*Ada*, cependant, il est possible de les implémenter très simplement depuis *Ada95* à l'aide d'objets protégés, que l'on pourrait assimiler à des moniteurs. Par exemple, une boîte à lettres de taille 1 avec écrasement peut être implémentée en *Ada95* de la façon suivante :

```
protected boîte_à_lettres is
      procedure déposer(delta : in donnée) ;
      -- Cette primitive est non bloquante car la boîte est à écrasement
      entry retirer(delta : out donnée) ;
```

```
      -- Cette primitive est bloquante, caractérisée par le mot clé
      -- entry à la place de procedure
private
      d : donnée ;
      vide : boolean :=true ;
      -- La boîte est initialement vide
end boîte_à_lettres ;
```

```
protected body boîte_à_lettres is
      procedure déposer(delta : in donnée) is
      begin
            d := delta ;
            vide := false ;
      end déposer ;
      entry retirer(delta : out donnée) when not vide is
      -- la tâche appelante est bloquée tant que la boîte est vide
      begin
            delta := d ;
            vide := true ;
      end retirer ;
end boîte_à_lettres ;
```

ii) Communication synchrone : le rendez-vous

La seconde forme de communication, initialement proposée dans *Ada 83* est la communication par **rendez-vous**. Ici, deux tâches se synchronisent pour s'échanger des données et éventuellement exécuter un code en commun. L'une des tâches, celle qui demande le rendez-vous, attend que l'autre tâche accepte le rendez-vous. Lorsque le rendez-vous est établi, la tâche qui a accepté le rendez-vous exécute une partie de code de façon synchronisée avec l'autre tâche. La tâche demandeur attend que le code soit terminé avant de continuer seule. Ce concept est rarement implémenté tel quel dans les noyaux temps réel car il peut être obtenu à l'aide de deux boîtes aux lettres pour chaque rendez-vous $\texttt{rdz-vous}_i$:

```
-- Lors de l'initialisation du système
      Créer_BAL(demande_rdz_vousi,1)
      Créer_BAL(accepter_rdz_vousi,1)
-- Demande de rendez-vous dans le corps de la tâche demandeur
      Déposer_BAL(demande_rdz_vousi,paramètres_du_rdz-vous)
      Retirer_BAL(accepter_rdz_vousi,paramètres_de_retour_du_rdz-vous)
      -- La tâche demandeur se met en attente de la fin du rendez-vous
-- Acceptation de rendez-vous dans le corps de la tâche acceptrice
      Retirer_BAL(demande_rdz_vousi,paramètres_du_rdz-vous)
      ...
      -- Code du rendez-vous
      Déposer_BAL(accepter_rdz_vousi,paramètres_de_retour_du_rdz-vous)
```

Les communications synchrones peuvent ainsi être obtenues à l'aide de primitives de communication asynchrone. Nous nous limiterons donc à l'étude de communications asynchrones.

iii) Synchronisations : Les signaux ou événements

Une alternative à la communication par boîte aux lettres est la communication par **signaux** (voir tableau I-1-3), on parle aussi d'**événements**. Des tâches peuvent à certains moments de la vie de l'application émettre ou attendre des événements. La différence majeure existant entre événements et messages est la possibilité d'envoyer ou d'attendre des conjonctions ou disjonctions d'événements. Par exemple une tâche peut attendre $(E_1\&E_2)|E_3$, c'est à dire que

les événements E_1 et E_2 soient émis ou bien que l'événement E_3 soit émis. Il y a de plus deux façons de signaler un événement : **mémorisé** ou non.

Instruction	Paramètres	Post-condition
Créer_Evt	ε : nom d'événement	un événement ε est créé. Une file d'attente vide de tâches en attente est créée. L'état de ε est « non signalé ».
Signaler_Evt	X : conjonction d'événements	les événements de X deviennent signalés. Toutes les tâches bloquées en attente dans les files de ces événements voient leur condition de réveil réévaluée.
Effacer_Evt	X : conjonction d'événements	l'état des événements de X devient non signalé.
Cliquer_Evt	X : nom d'événement	cette instruction équivaut à successivement signaler et effacer les événements de X. Toutes les tâches bloquées en attente dans les files des événements cliqués voient leur condition de réveil réévaluée. Cette instruction correspond à l'émission d'un événement non mémorisé.
Attendre_Evt	X : combinaison logique d'événements conjonctifs ou disjonctifs	si les événements attendus sont signalés, la tâche exécutant cette instruction continue normalement son exécution, sinon, la tâche passe dans l'état « bloquée » et est mise dans les files d'attente des événements de X.

tableau I-1-3 : primitives de gestion d'événements dans un noyau temps réel minimaliste

Ce type de communication est hautement utilisé dans les langages synchrones, mais beaucoup moins dans les langages asynchrones. D'ailleurs les gestions d'événements ne sont pas proposées par Ada. On peut imputer cela au fait que les systèmes utilisant ces primitives sont difficiles à étudier dès lors que l'on se trouve dans une hypothèse asynchrone. En effet, dans ce cas, le comportement des tâches est fortement influencé par l'ordonnancement (voir exemple I-1-1).

---

exemple I-1-1 : étude de deux tâches communicant via événements.

Soient deux tâches τ_1 et τ_2, telles que τ_1 clique l'événement E_1 et que τ_2 attende cet événement. Si l'instruction `Cliquer_Evt` de τ_1 est exécutée après l'instruction `Attendre_Evt` de τ_2, alors τ_2 peut continuer son exécution (voir figure I-1-5), sinon τ_2 reste en attente (voir figure I-1-6) jusqu'à ce que E_1 soit à nouveau signalé. Or si τ_1 et τ_2 sont actives en même temps, c'est la politique d'attribution des processeurs uniquement qui décide du sort de τ_2. On aboutit à la même remarque sur l'influence de l'ordonnancement pour les événements mémorisés : le problème se pose lorsque l'événement est effacé : si la tâche effaçant l'événement s'exécute avant une tâche qui est censée l'attendre, le comportement du système est différent du cas où l'ordre d'exécution est inversé.

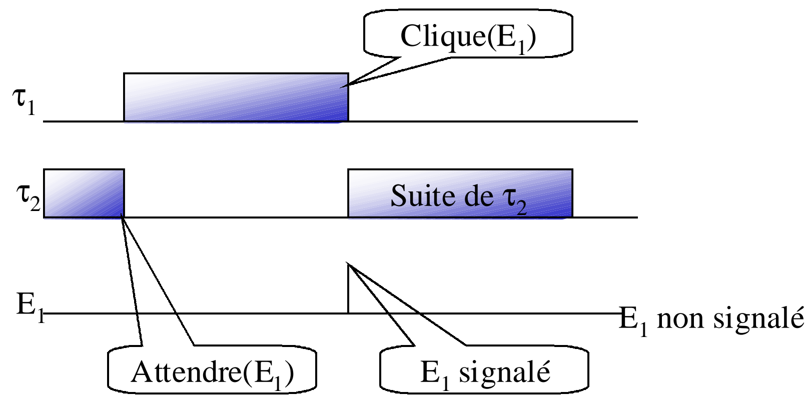


figure I-1-5 : événement non mémorisé : cas où l'événement a lieu après la mise en attente

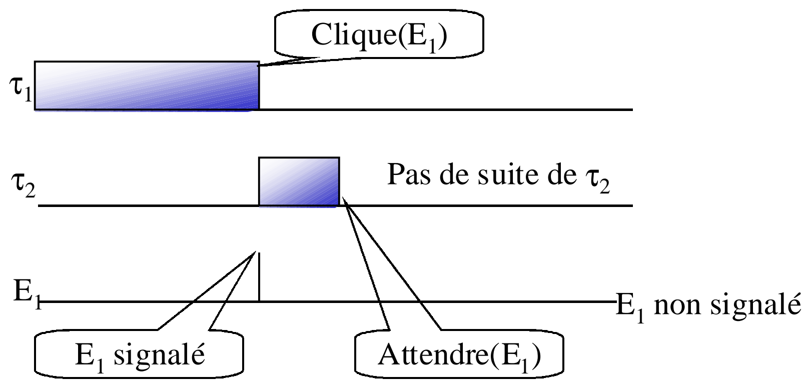


figure I-1-6 : événement non mémorisé : cas où l'évènement a lieu avant la mise en attente

L'interaction existant entre ordonnancement des tâches et fonctionnement des tâches communicant via événements justifie le fait que les événements ne sont généralement pas étudiés dans le cadre de systèmes temps réel asynchrones.

I-1.3.1.c Gestion des ressources critiques

Les ressources sont critiques si elles ne peuvent être utilisées simultanément que par un nombre restreint de tâches. Par exemple, certains périphériques, tels le terminal, une imprimante ou les stockages, sont souvent restreints à un seul accès simultané. De même, l'accès en écriture à une même zone mémoire ne peut être effectué que par une seule tâche. Le concept sous-jacent aux ressources critiques se nomme l'**exclusion mutuelle** : il s'agit d'assurer que l'accès à une ressource (on parle alors de **section critique**) ne peut pas être interrompu par un autre accès à cette même ressource. Cette notion, ainsi que la notion d'interblocage, est détaillée dans les chapitres 2.2 et 3.3 de [Tan 99].

Il existe plusieurs solutions au problème d'exclusion mutuelle, mais la solution la plus répandue dans les noyaux temps réel s'appuie sur les **sémaphores**.

Chaque ressource est protégée par un sémaphore, initialisé par le nombre d'accès simultanés autorisés sur la ressource. Lorsqu'une tâche souhaite utiliser une ressource, elle doit prendre le sémaphore, et le rendre lorsqu'elle n'a plus besoin de la ressource.

Il existe aussi des ressources en lecture/écriture, pour lesquelles *n* accès simultanés en lecture sont possibles, alors qu'un seul accès simultané en écriture n'est possible, excluant tout accès simultané en lecture. Ce modèle de ressources n'a été étudié que par peu d'auteurs,

comme Zhao, Ramamritham et Stankovic qui l'ont étudié dans [ZRS 87a][ZRS 87b]. Cependant, il est indispensable à l'étude de systèmes de tâches possédant des ressources en lecture ou en écriture : par exemple une tâche peut écrire la température dans un tampon, qui peut être lu simultanément par une tâche de contrôle et une tâche d'affichage. Ce type d'accès à des ressources peut être facilement implémenté à l'aide de sémaphores non-binaires[1] : le sémaphore est initialisé à n, nombre d'accès concurrents possibles en lecture. Un lecteur prend une unité du sémaphore, alors qu'un écrivain prend les n unités.

Le tableau I-1-4 représente les primitives de gestion de sémaphore classiquement proposées dans les noyaux temps réel.

Instruction	Paramètres	Post-condition
Créer_Sémaphore	σ : nom de sémaphore ν : nombre d'unités utilisables simultanément, ou bien nombre de lecteurs simultanés autorisés	un sémaphore σ est créé et initialisé à ν
Prendre_Sémaphore	σ : nom de sémaphore ν : nombre d'unités requises	la tâche exécutant cette instruction prend ν unités du sémaphore σ. Si le sémaphore possède suffisamment d'unités, (i.e. supérieur à ν), alors il est décrémenté de ν. Cette instruction peut être bloquante : si σ possède un nombre d'unités inférieur à ν, la tâche passe dans l'état « en attente » jusqu'à ce que ν unités soient disponibles
Rendre_Sémaphore	σ : nom de sémaphore ν : nombre d'unités rendues	la tâche exécutant cette instruction, qui devait avoir préalablement pris ν unités du sémaphore, les rend. C'est à dire que la valeur du sémaphore est incrémentée de ν.

tableau I-1-4 : primitives de gestion de ressources dans un noyau temps réel minimaliste

L'une des implémentations les plus répandues pour la gestion des sémaphores est la suivante : à chaque sémaphore σ, est associée une file *fifo* par niveau de priorité des tâches requiérant σ. Lorsque plusieurs tâches sont en attente sur un sémaphore, la tâche choisie est la première tâche de la file de plus haute priorité.

Les sémaphores ne sont pas visibles directement en *Ada*, mais ils peuvent être aisément implémentés à l'aide du concept de plus haut niveau de **moniteur** (objet protégé) en *Ada95*, qui de plus facilite grandement l'implémentation de ressources en lecture/écriture. Voici un exemple de moniteur *Ada95* permettant l'accès à une donnée en lecture ou en écriture :

```
protected ressource_lecture_ecriture is
      procedure écrire(delta : in donnée) ;
      -- modifie la donnée interne
      function lire return donnée ;
      -- renvoie la valeur de la donnée interne
private
```

[1] Sémaphore binaire : sémaphore ne pouvant prendre que les valeurs libre (1) et pris (0)

```
      d : donnée ;
end ressource_lecture_ecriture ;
```

```
protected body ressource_lecture_ecriture is
      procedure écrire(delta : in donnée) is
      begin
            d := delta ;
      end écrire ;
      function lire return donnée is
      begin
            return d ;
      end lire ;
end ressource_lecture_ecriture ;
```

L'objet protégé proposé par *Ada95* n'autorise une modification interne de l'objet que par une procédure ou une entrée du sémaphore : une fonction ne peut faire un accès qu'en lecture seule sur l'état interne de l’objet protégé. Le noyau *Ada95* permet donc la réentrance des fonctions d'un même objet les unes par rapport aux autres (i.e. plusieurs lectures peuvent avoir lieu simultanément). Cependant, une procédure ou une entrée, qui peut modifier l'état interne de l'objet, a lieu en exclusion mutuelle avec toute autre procédure, entrée ou fonction sur le moniteur (i.e. la modification de l'état interne d'un moniteur s'effectue de manière atomique par rapport à toute autre accès au moniteur).

Le choix d'implémenter les ressources en lecture/écriture dans *Ada95* montre que ce modèle de ressources est très utilisé par les concepteurs, et qu'il se doit d'être étudié en ordonnancement.

De façon empirique, certains concepteurs protègent complètement l’accès en exclusion mutuelle à une ressource par une tâche en la rendant non préemptible à l’aide du masquage des interruptions, noté ***CLI*** (pour *CLear Interrupts* en assembleur). La tâche peut à nouveau être interrompue après qu’elle ait démasqué les interruptions avec une instruction ***STI*** (pour *SeT Interrupts*).

I-1.3.1.d La gestion du temps

Le temps est soit géré en temps « classique » (souvent en millisecondes), soit en unités de temps noyau, qui peuvent correspondre, par exemple, à la fréquence d’activation de l’ordonnanceur. L’utilisation de ces unités de temps noyau, dont la taille est plus ou moins paramétrable, présente l’assurance que les points de préemption se situent à unités de temps entières.

Les primitives de gestion du temps offertes par les noyaux temps réel se limitent souvent à une primitive de délai, attente pendant un certain temps ou jusqu’à une certaine date.

Par contre, chaque instruction possible est documentée non seulement par sa sémantique, mais aussi par le temps qu’elle requiert en fonction de l'architecture matérielle sous-jacente.

Ce qui nous intéresse le plus, dans le cadre de ce travail, c’est l’ensemble des politiques d’ordonnancement que les noyaux proposent. Une politique d’ordonnancement détermine l’ordre dans lequel les processeurs traiteront les tâches. Certains noyaux proposent des politiques d’ordonnancement en-ligne, d’autres hors-ligne, enfin certains ne proposent pas de techniques d’ordonnancement (ou bien des politiques qui ne peuvent pas être qualifiées de techniques d’ordonnancement temps réel telles que *First In First Out* ou *Round Robin*).

Qu’ils soient en-ligne ou hors-ligne, les algorithmes d’ordonnancement reposent tous sur une connaissance du système de tâches réduites à leut comportement temporel et des interactions qui existent entre les tâches. Le paragraphe suivant traite donc de la modélisation qui est faite au niveau des systèmes de tâches.

I-1.3.2 Contextes temps réel

La grande diversité des domaines d'application du qualificatif temps réel impose une terminologie stricte définissant les contraintes temporelles, plus ou moins fortes, imposées aux systèmes temps réel.

- Systèmes temps réel à contraintes strictes (***TRCS***) : systèmes composés de tâches soumises à des contraintes temporelles strictes. Chaque occurrence de tâche se voit associer une échéance qu'il serait inadmissible de ne pas respecter. Une faute temporelle pourrait être coûteuse humainement et/ou financièrement. Les tâches composant de tels systèmes sont dites *TRCS*.
- Systèmes temps réel à contraintes relatives (***TRCR***) : systèmes composés de tâches contraintes temporellement par une échéance comme dans le cas des systèmes *TRCS*. Cependant, dans ce contexte, la violation d'une contrainte temporelle est tolérable bien qu'elle entraîne un coût que l'on cherche à minimiser. Les tâches composant de tels systèmes sont dites *TRCR*.
- Systèmes temps réel à contraintes mixtes (***TRCM***) : qualificatif regroupant la plupart des systèmes temps réel puisque les systèmes *TRCM* sont composés de tâches *TRCS* et de tâches *TRCR*. La problématique dans ce contexte est d'éliminer toute possibilité de faute temporelle pour les tâches *TRCS* tout en minimisant les fautes temporelles de tâches *TRCM*.

I-1.3.3 Tâches temps réel

Les tâches temps réel peuvent être périodiques, sporadiques ou apériodiques. D'autres tâches, appelées cycliques, sont étudiées en ordonnancement non temps réel. Les systèmes de tâches cycliques forment une classe de problèmes à part entière, que nous n'étudierons pas en détails dans cette thèse. Le lecteur intéressé par l'ordonnancement de tâches cycliques peut se référer à [HM 95] pour un survol. La modélisation par réseaux de Petri y est souvent utilisée, notamment dans [CC 88][Ric 97].

I-1.3.3.a Tâches périodiques

Une tâche τ_i est dite **périodique** si elle est activée à intervalles réguliers.

Le modèle classiquement utilisé pour représenter les tâches périodiques, attribué à Liu et Layland [LL 73], est présenté sur la figure I-1-7. Ce modèle, bien que ne représentant qu'un sous-ensemble très restreint des systèmes de tâches pouvant être implémentés dans un noyau temps réel, est très largement utilisé.

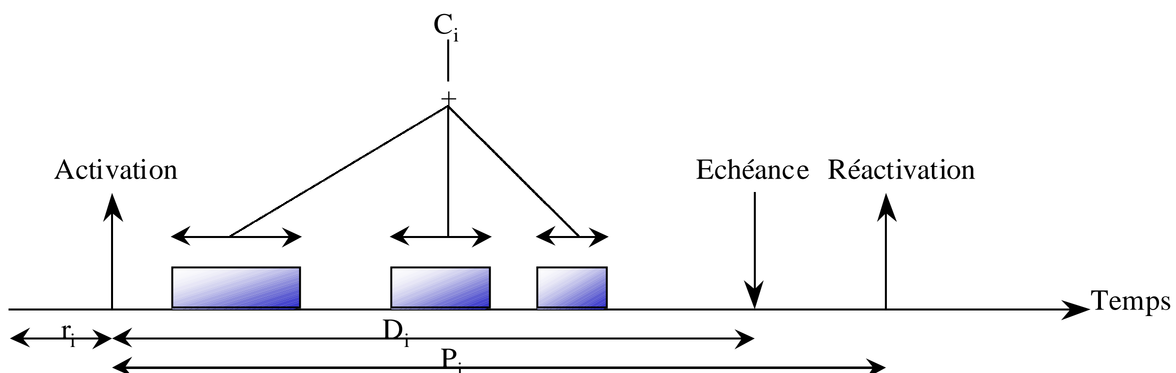


figure I-1-7 : modèle initial d'une tâche temps réel

Chaque tâche τ_i y est simplement modélisée par quatre paramètres temporels :

Sa **première date d'activation** (appelée aussi **date de réveil** ou ***offset***) $\boldsymbol{r_i}$. Lorsque toutes les tâches sont initialement réveillées au même instant $r_1=r_2=\ldots=r_n$, on dit que les tâches sont **simultanées** (on peut aussi trouver le terme synchrone dans la littérature) et sans perte de généralité on pose $r_1=r_2=\ldots=r_n=0$. Dans le cas contraire, on prend $min_{\tau_i\in S}\{r_i\}$ comme origine des temps, et toutes les dates de réveil sont décalées afin de conserver le même écart relatif entre chacune d'entre elles. Les tâches τ_i telles que $r_i>0$ sont dites **différées**.

Une **charge processeur** $\boldsymbol{C_i}$, qui est le temps processeur maximal requis par chaque exécution de τ_i. Il est important de noter que ce temps est une borne supérieure du temps d'exécution requis par une tâche. Cette borne est loin d'être triviale à obtenir [Shaw 89][PK 89][PS 91][PN 98], chapitre VIII de [Bab 96][BC 98].

Une **période d'activation** $\boldsymbol{P_i}$: τ_i est activée toutes les P_i unités de temps à partir de la date r_i. Il est donc aisé de calculer chaque date d'activation de τ_i à travers le temps puisqu'elle est activée aux dates $\boldsymbol{r_{i,k}}=r_i+(k-1)P_i \ \forall k\in \mathbb{N}^+$, $\mathbb{N}^+$ étant l'ensemble des entiers positifs : on dit que le réveil de la $k^{\text{ième}}$ **occurrence** $\tau_{i,k}$ (nommée aussi **requête** ou **instance**) de τ_i a lieu à la date $r_{i,k}$.

Un **délai critique** $\boldsymbol{D_i}$, temps alloué à la tâche pour se terminer après chacune de ses activations. L'**échéance** $\boldsymbol{d_{i,k}}$ de la $k^{ième}$ occurrence de τ_i se calcule à l'aide du délai critique. En effet à chaque date $d_{i,k}=r_{i,k}+D_i \ \forall k\in \mathbb{N}^+$, $\tau_{i,k}$ voit son échéance choir et doit être terminée. Si ce n'est pas le cas, il se produit une faute temporelle.

Une tâche est dénotée $\tau_i = \langle r_i,C_i,D_i,P_i\rangle$, et un système de tâches $S = \{\langle r_1,C_1,D_1,P_1\rangle,\ldots, \langle r_n,C_n,D_n,P_n\rangle\}$.

Puisque C_i/P_i est la fraction d'un processeur dont τ_i a besoin pour s'exécuter, la fraction processeur nécessitée par le système de tâches S est donné par $U_S=\sum_{i=1}^{n} C_i/P_i$, n étant le nombre de tâches de S. U_S est la **charge** (ou **taux d'utilisation**) du système de tâches S. Si U_S est plus grand qu'un entier p, alors le système ne peut se contenter de p processeurs. Par contre, si le nombre de processeurs est p et que $U_S < p$, alors les processeurs ne seront pas occupés pleinement par le système, on dit qu'ils seront périodiquement **oisifs**.

La Définition I-1-3 regroupe les notations définies ci–dessus.

Définition I-1-3 : paramètres temporels des tâches temps réel

- Date de réveil r_i
- Charge maximale C_i
- Délai critique D_i
- Période d'activation P_i

Les paramètres suivants sont déduits des 4 précédents :

- $k^{\text{ième}}$ activation $r_{i,k} = r_i+(k-1)P_i$, correspond au réveil de la $k^{\text{ème}}$ occurrence $\tau_{i,k}$ de τ_i
- $k^{\text{ième}}$ échéance $d_{i,k} = r_{i,k}+D_i$
- Charge processeur d'une tâche $U_i = C_i/P_i$
- Charge processeur d'un système $U_S = \sum_{\tau_i\in S} U_i$

La figure I-1-8 reprend les notations de la définition précédente.

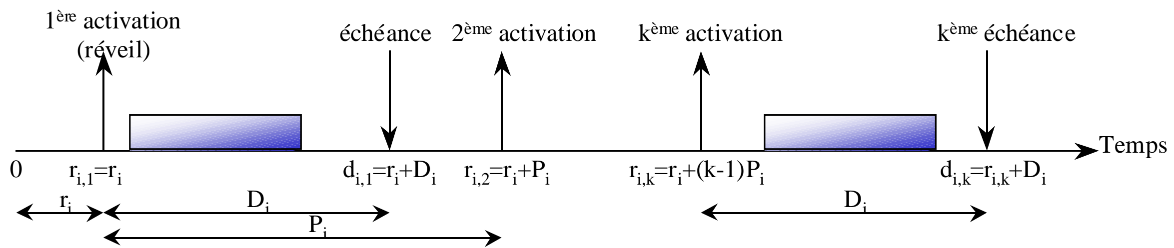


figure I-1-8 : chronogramme des activations et échéances d'une tâche périodique

Généralement, le délai critique est choisi plus petit que la période afin d'interdire toute **réentrance**[2] d'une tâche. Cela est dû au fait que, bien que théoriquement l'occurrence d'une tâche soit déclenchée par l'horloge temps réel (*HTR*), les tâches sont rendues périodiques dans les langages de programmation temps réel par l'emploi d'une boucle contenant une instruction de mise en sommeil jusqu'à une date donnée. A chaque itération de cette boucle, la tâche s'endort, puis est réveillée par l'*HTR*. C'est donc toujours le même code qui est exécuté, pour chaque occurrence d'une tâche. Permettre la réentrance d'une tâche reviendrait à autoriser plusieurs pointeurs d'instructions sur le même code, ce qui est envisageable, mais surtout le partage de variables locales à une tâche, ce qui impliquerait la recopie complète des données internes d'une tâche dans chacune de ses occurrences.

De par la contrainte de non réentrance des tâches périodiques, le respect de la période entraîne un délai critique maximum qui est la période de la tâche elle-même (on parle alors de tâches à **échéance sur requête**).

Les paramètres temporels définissent de façon statique les contraintes temporelles d'un système de tâches. Lorsque celui-ci est exécuté, des caractérisations dynamiques servent à rendre compte de la qualité temporelle de l'application mise en œuvre. Ceux-ci dépendent de la technique d'ordonnancement mise en œuvre.

Définition I-1-4 : Caractérisations de l'exécution d'un système de tâches (voir figure I-1-9).

- Le début de $\tau_{i,k}$, noté ***début$_{i,k}$***, est la date à laquelle $\tau_{i,k}$ dispose pour la première fois du processeur.
- La terminaison de $\tau_{i,k}$, notée ***terminaison$_{i,k}$***, est la date à laquelle $\tau_{i,k}$ est terminée.
- Le **temps de réponse** de $\tau_{i,k}$, noté ***TR$_{i,k}$***, est donné par $TR_{i,k}=terminaison_{i,k}-r_{i,k}$. Par définition, une faute temporelle a lieu si et seulement si le temps de réponse d'une occurrence dépasse son délai critique.
- La **latence** (ou **temps de retard admissible**) de $\tau_{i,k}$ est donnée par ***latence$_{i,k}$***$=d_{i,k}-terminaison_{i,k}$.

[2] Une tâche est dite réentrante si il est possible que plusieurs de ses occurrences soient en cours d'exécution simultanément

– La **laxité**[3] de $\tau_{i,k}$ est donnée à l'instant t par sa marge de manœuvre, c'est à dire $laxité_{i,k}(t)=d_{i,k}-t-C_{i,k}(t)$ où $C_{i,k}(t)$ est le nombre d'unités de temps de $\tau_{i,k}$ restant à traiter au temps t pour la terminer.

– Le **taux de réaction** de $\tau_{i,k}$ est donnée par $\boldsymbol{TxR_{i,k}}=\frac{TR_{i,k}}{D_i}$.

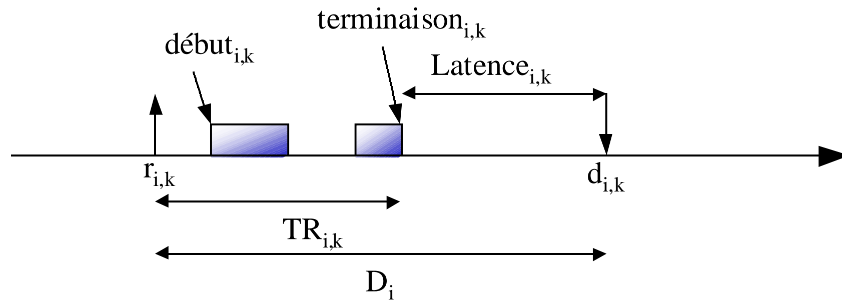


figure I-1-9 : notations caractérisant l'exécution d'une tâche

Certaines tâches peuvent être soumises à des contraintes de régularité, notamment les tâches d'acquisition. Cela est dû au théorème de *Shannon*[4].

Le lecteur pourra se référer à [Cot 97] pour plus de détails sur l'échantillonnage et le traitement du signal. Il faut retenir le fait que bien que les tâches soient périodiquement contraintes, elles ne sont pas forcément régulières, il est donc utile de caractériser la régularité des tâches.

Définition I-1-5 : Soit τ_i une tâche temps réel, sa **gigue temporelle** (on parle aussi de ***jitter***) est donnée par $gigue_i = max_{k\in \mathbb{N}^+}\{|(TR_{i,k+1}-TR_{i,k})|/D_i\}$. On parle de gigue nulle lorsque tous les temps de réponse de τ_i sont identiques.

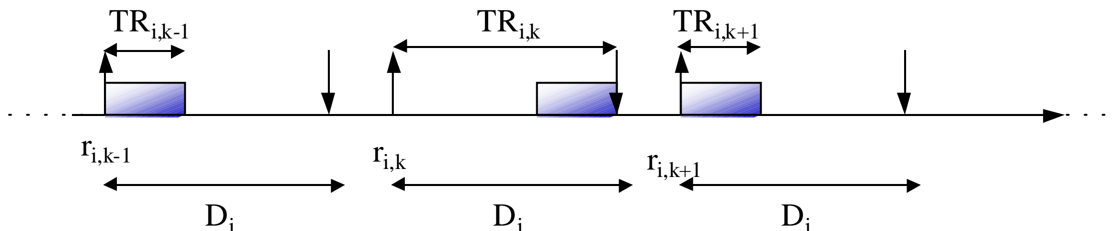


figure I-1-10 : exemple de gigue temporelle d'une tâche

La figure I-1-10 montre trois occurrences successives d'une tâche τ_i avec une gigue maximale $\frac{D_i - C_i}{D_i}$. Si dans un ordonnancement, une gigue nulle est assurée, la période d'une tâche

[3] On peut aussi définir la laxité comme étant la date au plus tard à laquelle on peut commencer à traiter les unités de temps restante d'une occurrence tâche [DM 89].

[4] Théorème de *Shannon* : l'échantillonnage d'un signal peut se faire sans perte d'information à une fréquence régulière deux fois supérieure à la fréquence du signal.

d'échantillonnage sera *1/2N* avec *N* fréquence du signal à échantillonner. Dans le cas contraire, la période de la tâche d'acquisition devra être réduite afin de prendre en compte la gigue pouvant être imposée à la tâche : on parle alors de **sur-échantillonnage**.

Les paramètres temporels des tâches sont obtenus à partir du cahier des charges du système. Généralement ils ne sont pas immuables, et certaines études ont montré qu'il pouvait être intéressant de faire varier certains d'entre eux afin, par exemple, de diminuer la gigue temporelle de certaines tâches [CCH 98].

Il est à noter que les paramètres temporels des tâches permettent de valider le comportement temporel du système et non pas son comportement fonctionnel. En effet, chaque instruction est abstraite à sa durée. Afin de prendre en compte les interactions entre les tâches telles qu'elles peuvent être créées à l'aide des primitives temps réel décrites en section I-1.3.1, nous définissons les contraintes de précédence et d'exclusion entre les tâches.

Lorsque les tâches ne sont définies que par leurs paramètres temporels, on dit qu'elles sont **indépendantes**. Lorsque les tâches communiquent, des **contraintes de précédence** sont induites par les interactions entre tâches. En effet, il est primordial de prendre en compte le fait qu'une tâche réceptrice soit en attente tant que la tâche émettrice associée n'a pas produit un message.

C'est d'ailleurs la prise en compte de contraintes de précédence qui justifie souvent l'existence de dates de réveil [Bla 76a][CSB 90]. Les contraintes de précédence étant prises en compte par notre modèle, leur existence peut se justifier dans l'exemple suivant [ATB 93] : supposons qu'une tâche doive lire périodiquement des données sur un disque dur, dont chaque temps d'accès est borné par 12 ms, puis envoyer le résultat à une tâche de traitement de même période. Il est intéressant de retarder la tâche de traitement de 12 ms par rapport à la tâche de lecture, même si la contrainte de précédence entre les tâches n'est pas éliminée par cet offset.

De plus, la plupart des applications temps réel nécessitent la prise en compte de **ressources critiques** (voir section I-1.3.1.c). Ces ressources impliquent certaines exclusions mutuelles, qui peuvent être, en lecture ou en écriture, nous avons vu aussi en section I-1.3.1.c qu'il était intéressant de pouvoir rendre certaines parties de tâches totalement non préemptibles à l'aide de l'instruction *CLI*.

I-1.3.3.b Tâches non périodiques

i) Tâches sporadiques

Les **tâches sporadiques**[KN 80][Mok 83] sont très utilisées dans les approches asynchrones du temps réel. Ce sont des tâches dont on ne connaît pas a priori la période, mais dont on connaît l'intervalle minimal séparant deux requêtes successives. Une tâche sporadique τ_{spor} peut être caractérisée par trois paramètres temporels :

– C_{spor} le temps processeur nécessaire au traitement de chaque occurrence de τ_{spor},
– D_{spor} le délai critique,
– P_{spor} l'intervalle minimal séparant deux occurrences de τ_{spor}.

La figure I-1-10 montre que la $k^{ème}$ occurrence d'une tâche sporadique τ_i, est caractérisée temporellement par sa date d'occurrence $r_{i,k}$ et son échéance $d_{i,k}=r_{i,k}+D_i$.

La difficulté de validation de systèmes de tâches sporadiques réside dans le fait que les dates $r_{i,k}$ des requêtes sont inconnues a priori.

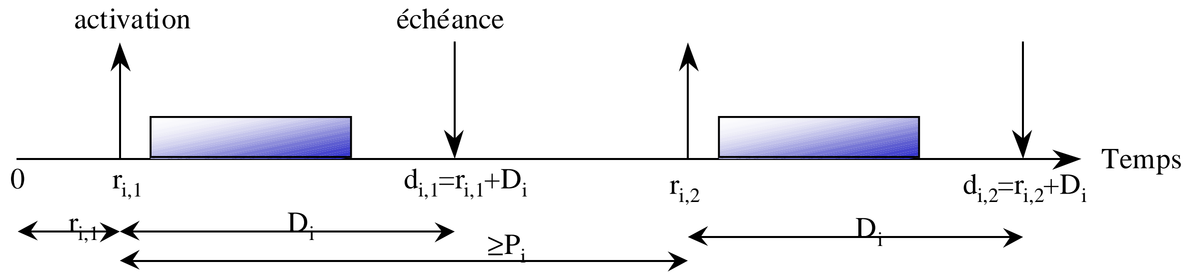


figure I-1-11 : chronogramme d'une tâche sporadique

Les tâches sporadiques à contraintes strictes ne peuvent être garanties sans concession. En effet, puisque les requêtes sporadiques ne sont pas connues a priori, le seul moyen de garantir que toute occurrence d'une sporadique sera traitée à temps est de le faire dans le pire des cas. C'est à dire que le cas validé est celui dans lequel la tâche sporadique est déclenchée le plus souvent possible. Il ne s'agit pas de demander au concepteur ce qu'est le plus souvent possible, mais de garantir que quelle que soit la date à laquelle une occurrence de tâche sporadique apparaît, une place dans l'ordonnancement est prévue pour elle. Cela peut être fait en représentant la tâche sporadique τ_{spor} de délai critique D_{spor} par une tâche **serveur périodique de sporadique** τ_{serv} choisie telle que :

- son temps de traitement soit assez grand pour pouvoir y placer τ_s, i.e. $C_{serv} \geq C_{spor}$
- quelle que soit la date d'occurrence de τ_{spor}, elle puisse être traitée dans le temps imparti à τ_{serv}, i.e. $P_{serv}+D_{serv} \leq D_{spor}$. Par exemple, sur la figure I-1-12, une requête de tâche sporadique a lieu $\alpha \geq 0$ unités de temps trop tard pour être traitée par la $k^{ème}$ occurrence de τ_{serv}, elle n'est donc traitée que par l'instance suivante du serveur.

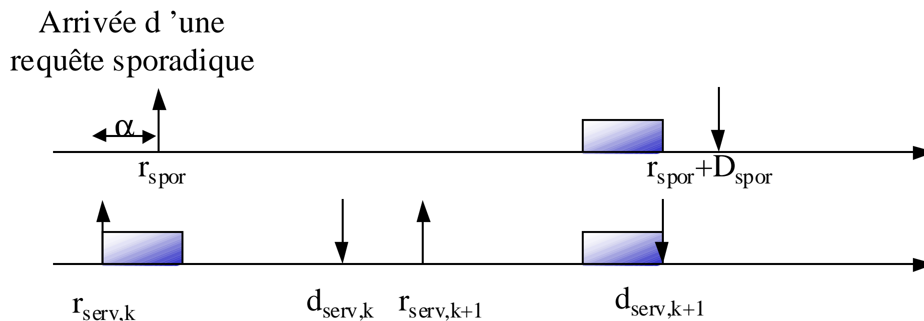


figure I-1-12 : traitement d'une requête de tâche sporadique dans le serveur périodique associé

Afin de répondre à ces deux contraintes tout en minimisant l'augmentation de la charge du système et en maintenant l'ordonnançabilité du système, les valeurs limites sont choisies, c'est à dire $C_{serv}=C_{spor}$, et le couple (D_{serv},P_{serv}) est choisi tel que $P_{serv}+D_{serv}=D_{spor}$. La figure I-1-13 montre les choix possibles pour le couple (D_{serv},P_{serv}) tout en respectant la condition nécessaire d'ordonnançabilité $C_{serv} \leq D_{serv}$.

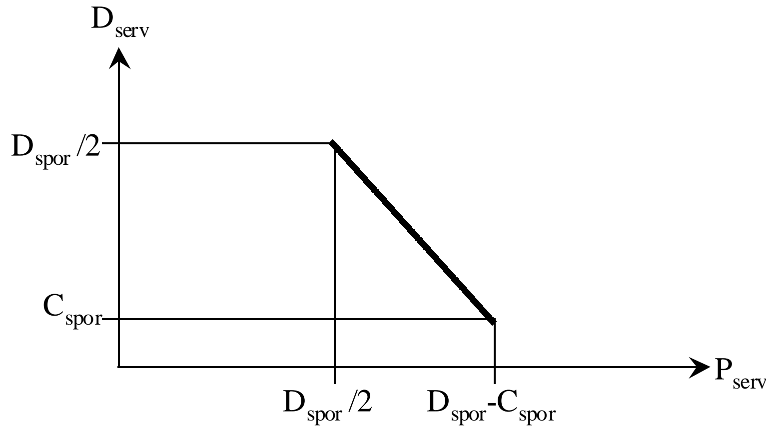


figure I-1-13 : valeurs limites des paramètres temporels d'un serveur périodique de tâche sporadique

Dans la littérature, le couple (D_{serv},P_{serv}) choisi est souvent $(D_{spor}/2,D_{spor}/2)$. Bien que ce choix maximise la charge processeur par rapport aux autres choix possibles, c'est lui qui est le moins contraignant dans les conditions suffisantes d'ordonnançabilité que nous donnerons par la suite. Cependant, dans le cadre de l'ordonnancement exhaustif, d'autres valeurs de couples peuvent être choisies sans nuire à l'ordonnançabilité du système.

ii) Tâches apériodiques

Les tâches **apériodiques** sont des tâches pour lesquelles le délai minimal entre deux occurrences est inconnu. On ne peut donc valider a priori un système composé d'apériodiques à contraintes strictes. Les apériodiques sont de ce fait considérées à contraintes relatives et sont prises en compte dans le cadre de systèmes *TRCM*. Elles sont soit prises en compte dans le **travail de fond**, c'est à dire dans les temps laissés libres par les tâches à contraintes strictes, soit prises en compte dans des **serveurs d'apériodiques**.

I-1.3.3.c Contextes d'ordonnancement

Afin de définir au mieux les problèmes d'ordonnancement et les solutions proposées dans certains cas particuliers, les **contextes d'étude d'ordonnançabilité** sont définis de la manière suivante :

χ=|*architecture* |*réveil*, *charge*, *délai critique*, *période*, *préemption, précédences*, *ressources*|

Un contexte est une suite d'éléments que nous nommons termes. Chacun des termes du contexte et ses valeurs possibles sont décrits dans le tableau I-1-5.

Terme	Valeurs possibles	Sémantique
architecture	p=1:m=1 p:m=1 p=1:m p:m	Un système monoprocesseur centralisé Un système multiprocesseur centralisé m systèmes monoprocesseurs répartis[5] m systèmes multiprocesseurs répartis
réveil	$r_i=0$ r_i	Tâches simultanées Certaines tâches sont différées
charge	$C_i=1$ C_i	Tâches de durée unitaire Tâches de durée quelconque

[5] Les systèmes répartis requièrent le prise en compte des caractéristiques du réseau utilisé. Ce cas n'étant pas traité par le suite, nous ne détaillons pas ici les architectures réseau.

délai critique	$D_i=P_i$ D_i	Tâches à échéance sur requête Tâches de délai critique quelconque
période	P_i ∅	Tâches périodiques Tâches non périodiques
préemption	prmpt (ou ∅) noprmpt prmptnoprmpt	Tâches préemptibles Tâches non préemptibles Des tâches ont des portions de code non préemptibles
précédences	∅ précformenormale préc	Pas de précédences Précédences en forme normale[6] Précédences quelconques
ressources	∅ res resRW	Pas de ressources critiques Ressources en exclusion mutuelle (§I-1.3.1.c) Ressources en lecture écriture

tableau I-1-5 : termes d'un contexte d'étude d'ordonnançabilité

Avec cette notation, $\chi=|p=1,m=1|r_i=0,C_i,D_i=P_i,P_i|$ dénote l'étude des systèmes de tâches indépendantes préemptibles périodiques simultanées, de charge quelconque, à échéance sur requête, en environnement monoprocesseur. La valeur de p dénote le nombre de processeurs par site, alors que m dénote le nombre de sites. Dans le contexte monoprocesseur centralisé (i.e. $p=1$ et $m=1$), on écrit de façon plus brève $|1|r_i=0,C_i,D_i=P_i,P_i|$. Les contextes les plus larges que nous pouvons envisager de décrire à l'aide de cette syntaxe sont donc les systèmes de tâches quelconques partageant des ressources en lecture et écriture, pouvant avoir des parties non préemptibles, des contraintes de précédence quelconques, sur une architecture matérielle composée de multiprocesseurs répartis $|p,m|r_i,C_i,D_i,P_i,$prmptnoprmpt,prec,resRW$|$.

Cette notation s'inspire du formalisme $|\alpha|\beta|\gamma|$ introduit dans [GLLR 79] pour définir les problèmes d'ordonnancement classique, et utilisé dans un contexte de placement temps réel dans [Bea 96]. α symbolise la configuration matérielle, β la configuration fonctionnelle, et γ décrit les contraintes et les objectifs à atteindre. Dans un contexte d'ordonnancement temps réel, γ contient au moins le respect des contraintes temporelles des tâches, on ne représente donc pas cette contrainte.

Soit la relation d'ordre *est moins général que*, notée $\subset$, définie sur α par un treillis représenté sur la figure I-1-14, et sur les termes de β dans le Tableau I-1-6.

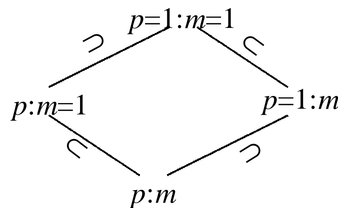


figure I-1-14 : relation d'ordre définie sur la configuration matérielle du contexte

Relation d'ordre	Explication
$r_i=0\subset r_i$	Les tâches simultanées sont un cas particulier des tâches différées
$Ci=1\subset Ci$	Les tâches de charge unitaire sont un cas particulier des

[6] Les contraintes de précédence entre des tâches sont dites en forme normale si les attentes de message se trouvent en début de tâche, et les émissions en fin de tâche (voir section I-2.2.2)

	tâches de durée quelconque
$D_i=P_i \subset D_i$	Les tâches à échéance sur requête sont un cas particulier des tâches temps réel
$P_i \subset \emptyset$	Les tâches périodiques peuvent être ramenées à des tâches non périodiques à un dépliage (découpage des tâches périodiques en non périodiques sur une durée de simulation que nous expliciterons en section III) près. Nous verrons cela plus en détail lorsque nous parlerons de la durée de simulation. Cependant, le dépliage n'est pas polynomial
prmpt⊂noprmpt⊂prmptnoprmpt	Les tâches préemptibles sont un cas particulier des tâches totalement non préemptibles qui sont elles-même un cas particulier des tâches mixtes
∅⊂précformenormale⊂préc	Les tâches sans contraintes de précédence sont un cas particulier des tâches dont les contraintes de précédence peuvent être mises sous forme normale, qui sont un cas particulier des contraintes de précédence quelconques
∅⊂res⊂resRW	Les tâches sans ressources sont un cas particulier des tâches soumises à des contraintes de ressources en exclusion mutuelle, qui sont un cas particulier des ressources pouvant être accédées en lecture ou en écritue

Tableau I-1-6 :relation d'ordre sur les termes d'un contexte temps réel

La relation d'ordre ⊂ est généralisée à un contexte temps réel sous la forme d'un treillis, avec $\chi_1=|\alpha_1|\beta_{1,1},\beta_{1,2},\ldots| \subseteq \chi_2=|\alpha_2|\beta_{2,1},\beta_{2,2},\ldots|$ si et seulement si $\alpha_1 \subseteq \alpha_2$ et $\beta_{1,1} \subseteq \beta_{2,1}$, $\beta_{1,2} \subseteq \beta_{2,2}\ldots$

Cette relation d'ordre exprime le fait qu'un résultat présenté pour un contexte donné χ s'applique à tout contexte χ' tel que $\chi' \subseteq \chi$. Il faut cependant garder en mémoire que si ce résultat concerne la complexité algorithmique, le passage d'un système de tâches périodiques à son dépliage en tâches non périodiques n'est pas polynomial.

I-1.4 Conclusion

Sous l'hypothèse asynchrone, il est nécessaire de programmer les applications temps réel sous la forme d'un ensemble de tâches, qui peuvent communiquer, partager des ressources, et avoir des parties non préemptibles. Les systèmes temps réel étant le plus souvent dédiés au contrôle de procédés dont la durée de vie n'est pas définie clairement, on considère généralement que leur durée de vie est illimitée. Par conséquent, ils sont composés de tâches périodiques. Afin de faciliter leur étude, les tâches sporadiques sont transformées en tâches périodiques représentant leur fonctionnement au pire, et les tâches apériodiques sont prises en compte par des tâches périodiques appelées serveurs d'apériodiques. Le problème est que pour ce type de tâche, les contraintes strictes ne peuvent être gérées étant donné qu'aucune information sur l'intervalle minimal séparant deux occurrences de ces tâches n'est disponible.

La plupart des études de systèmes temps réel à contraintes strictes s'intéressent donc à des ensembles de tâches périodiques, définies par leur période, leur délai critique, leur date de réveil, et leur corps. Ce dernier est abstrait à la durée des instructions et aux primitives temps réel (interactions possibles avec d'autres tâches) qui le composent.

Cette partie présente un état de l'art en ordonnancement temps réel. Après quelques définitions, nous présentons les deux grands courants d'ordonnancement temps réel : le « en-ligne », et le « hors-ligne ». Nous développons les approches en-ligne classiques afin d'en expliquer les avantages, notamment lorsque les tâches sont à tout moment préemptibles, mais aussi afin d'en exprimer les limitations lorsque les tâches utilisent des ressources critiques. Enfin, une étude des algorithmes hors-ligne les plus connus permettra de voir sous quelles hypothèses ils sont utilisables, et quels sont leurs objectifs.

I-2 L'ordonnancement des systèmes de tâches temps réel

La section I-1.2.2 justifie l'intérêt d'utiliser des tâches. De plus, un système temps réel ayant une durée de vie indéterminée, et son étude devant se ramener à une durée bornée, les tâches qui le composent doivent nécessairement être périodiques pour une étude systématique. Nous avons vu en section I-1.3.3.b comment prendre en compte les tâches sporadiques dans le modèle périodique.

I-2.1 Propriétés générales des algorithmes d'ordonnancement

I-2.1.1 Notions de validité, de puissance, et d'optimalité

Le temps est découpé en quanta uniformes, pendant lesquels au plus une tâche est affectée à un processeur. Une séquence est une suite de placements, un placement $(\tau_{\sigma,t,1},\ldots,\tau_{\sigma,t,p})$ signifiant que l'occurrence courante à l'instant t de la tâche $\tau_{\sigma,t,k}\in S$ est affectée au le processeur k pendant l'intervalle $[t..t+1[$. Etant donné que la charge d'un système de tâches n'est pas forcément suffisante pour occuper tout le temps tous les processeurs, il arrive que des processeurs soient **inactifs**, on dit aussi **oisifs.** Nous verrons qu'une partie de ces **temps d'inactivité**, appelés aussi **temps creux**, sont prévisibles et peuvent être englobés dans une tâche périodique nommée **tâche oisive** et notée τ_0, et que l'autre partie des temps creux, qui sont en nombre borné pour les ordonnancements valides, peut être prise en compte par une tâche non périodique, appelée **tâche creuse**, et dénotée τ_c.

Définition I-2-1 : Soit $S=\{\tau_i\}_{i=1..n}$ un système de tâches donné dans un contexte $\chi=|architecture|réveil, charge, délai\ critique, période, préemption, précédences, ressources|$. Une **séquence** d'ordonnancement $\sigma=(placement_1, placement_2,\ldots)$, avec $placement_t=(\tau_{\sigma,t,1},\ldots,\tau_{\sigma,t,p})$, avec p nombre de processeurs, où $\tau_{\sigma,t,i}\in S\cup\{\tau_0,\tau_c\}$ pour $i=1,\ldots,p$, est l'affectation réalisée à la date t, est **valide** si et seulement si :

- Toutes les échéances tombées entre 0 et $|\sigma|$ sont respectées,
- Les contraintes de réveil, de périodicité, de précédence, d'exclusion mutuelle, et de non préemption sont respectées,
- Un placement contient au plus une occurrence de τ_i pour $i=1..n$, et peut contenir plusieurs occurrences de la tâche oisive τ_0 ou de la tâche creuse τ_c que nous définirons (ces tâches

représentent les temps creux des processeurs). Cette contrainte signifie qu'une tâche ne peut pas être traitée simultanément par plusieurs processeurs.

Les deux premiers points de la définition sont intrinsèques aux contraintes temporelles imposées aux tâches, et le troisième point indique que le code d'une tâche ne peut être affecté à plusieurs processeurs simultanément. Comme plusieurs processeurs peuvent être inactifs simultanément, plusieurs occurrences de τ_0 et τ_c peuvent apparaître dans une affectation. Par exemple, si il y a un seul processeur, une séquence d'ordonnancement peut être de la forme $((\tau_i), (\tau_j), (\tau_k), (\tau_i))$ signifiant que le processeur traite successivement pendant un quantum de temps à chaque fois τ_i, puis τ_j, puis τ_k, et enfin τ_i. Si il y a deux processeurs, la séquence $((\tau_i,\tau_0), (\tau_j,\tau_i), (\tau_j,\tau_i), (\tau_0,\tau_0))$ signifie que pendant le premier quantum de temps, le processeur 1 traite τ_i, alors que le processeur *2* ne fait rien. Pendant le deuxième quantum de temps, le processeur 1 traite τ_j, et le processeur *2* traite τ_i. Idem pendant le *3ème* quantum. Enfin, lors du *4ème* quantum de temps, les deux processeurs sont oisifs. La Définition I-2-1 interdit un placement (τ_i,τ_i) pour $\tau_i \in S$ puisque les tâches sont supposées non parallélisables.

Définition I-2-2 : Un système de tâches temps réel *S* est **ordonnançable** si et seulement si il existe une séquence d'ordonnancement de longueur infinie valide pour *S*.

Définition I-2-3 : Soit *A* un algorithme d'ordonnancement, et *S* un système de tâches temps réel. *S* est **fiablement ordonnancé** par *A* si et seulement si la séquence produite par *A* est valide pour *S*.

Les algorithmes d'ordonnancement peuvent être classés suivant leur puissance d'ordonnançabilité.

Définition I-2-4 : Un algorithme d'ordonnancement A_1 est **plus puissant** qu'un algorithme A_2 dans le contexte χ, noté $A_1 \geq_\chi A_2$, si et seulement si tout système de tâche dans le contexte χ fiablement ordonnancé par A_2 est fiablement ordonnancé par A_1.

L'opérateur $\geq_\chi$ définit un ordre partiel sur les algorithmes d'ordonnancement. Il reste maintenant à définir des maxima pour cet ordre partiel.

Définition I-2-5 : Soit un algorithme d'ordonnancement *A* et un contexte d'ordonnancement χ. On dit que *A* est un **algorithme d'ordonnancement optimal** pour χ si et seulement si pour tout algorithme *A'*, $A \geq_\chi A'$.

Cette définition s'étend aux classes d'algorithmes d'ordonnancement que nous verrons par la suite :

Définition I-2-6 : Soit un algorithme d'ordonnancement *A*, un contexte d'ordonnancement χ, et un ensemble (on dit aussi une classe) d'algorithmes d'ordonnancement X. On dit que *A* est un algorithme d'ordonnancement optimal dans la classe[7] des algorithmes de la classe X pour le contexte χ si et seulement si pour tout algorithme *A'* de X, $A \geq_{\chi} A'$.

Si tous les paramètres temporels de toutes les occurrences de tâches sont connus a priori, et que l'algorithme d'ordonnancement en tire profit pour ses décisions, l'algorithme est dit **statique**.

Dans le cas où l'algorithme d'ordonnancement ne connaît que les paramètres temporels des tâches actives, il est dit **dynamique**. On peut remarquer que pour les approches dynamiques, le fait de savoir que les tâches sont périodiques n'influe guère sur le comportement des algorithmes d'ordonnancement.

Bien qu'ils soient souvent confrontés à des systèmes de tâches périodiques dans un contexte temps réel, les algorithmes dynamiques, par définition, ne prennent pas en compte les périodes des tâches. En effet, dans le cas contraire, ils seraient capable de déduire les futures dates de réveil des tâches. Nous verrons que malgré cela, ils ont des propriétés d'optimalité intéressantes dans le cas monoprocesseur pour les configurations de tâches indépendantes.

Enfin, il convient de définir une classe d'algorithme d'ordonnancement qui nous permettra par la suite de définir des résultats très généraux :

Définition I-2-7 : Un **algorithme conservatif**, appelé aussi **au plus tôt**, est un algorithme d'ordonnancement ne prenant jamais la décision de ne rien faire lorsqu'au moins une tâche est active et non bloquée.

I-2.1.2 Validation temporelle de systèmes temps réel

La validation temporelle des systèmes temps réel repose sur deux principes :

- Le choix de l'ordonnancement, c'est à dire le choix d'un algorithme affectant à chaque instant les tâches aux processeurs,
- La preuve que le système est fiablement ordonnancé par l'algorithme choisi.

La preuve de l'ordonnançabilité d'un système par l'algorithme d'ordonnancement choisi nécessite la connaissance complète du système de tâches *TRCS*, ou au moins une borne supérieure à leur charge pour certains couples particuliers algorithme-contexte temps réel.

Dans tous les cas, une étude d'ordonnançabilité est effectuée avant la mise en œuvre d'un système temps réel. Deux façons de procéder sont utilisées :

- Génération d'une séquence d'ordonnancement valide (à répéter à l'infini) par l'algorithme choisi et implantation de la séquence dans une table qui sera consultée par un ordonnanceur simplifié que l'on appelle **séquenceur**. On parle d'**algorithme d'ordonnancement hors-ligne** puisque l'algorithme est exécuté avant.
- Choix d'un algorithme qui sera implanté dans l'ordonnanceur : on parle d'**algorithme d'ordonnancement en-ligne**. A chaque instant (éventuellement uniquement à chaque

[7] Une classe d'algorithme est définie par une propriété particulière telle que le fait qu'il soit conservatif ou non, à priorités statiques ou variables,…

réveil ou fin de tâche), l'algorithme en-ligne décide d'une affectation des tâches sur les processeurs. Cette décision possède bien entendu un coût sensiblement plus élevé que la décision prise par un séquenceur, ce coût qui n'est pas inhérent aux tâches temps réel mais à l'ordonnanceur est appelé **surcoût**.

Nous discutons dans les paragraphes suivants des avantages et inconvénients des différentes approches.

I-2.1.2.a Approches en-ligne

L'inconvénient majeur des algorithmes en-ligne est le surcoût imposé par le calcul des affectations des tâches aux processeurs par l'ordonnanceur. Cela implique une limitation de complexité dans le choix des algorithmes en-ligne : ils sont en général de complexité linéaire par rapport au nombre de tâches. Il faut préciser que la complexité des algorithmes en-ligne représente la complexité par unité de temps (ou par réveil ou fin de tâche) inhérente au choix des tâches à affecter aux processeurs.

La validation d'un algorithme en-ligne, toujours effectuée hors-ligne en ce qui concerne les tâches *TRCS*, s'effectue soit à l'aide de conditions analytiques basées sur les critères temporels des tâches, soit par simulation du système des tâches pendant une durée suffisante, appelée **durée de simulation**.

Les critères analytiques utilisés sont soit des conditions nécessaires et suffisantes d'ordonnançabilité, soit des conditions suffisantes. Il est important de souligner qu'un critère analytique s'applique dans un contexte restreint et pour un algorithme d'ordonnancement particulier. Si seule une condition suffisante (**CS**) est disponible dans le contexte considéré pour l'algorithme donné, et que celle-ci n'est pas vérifiée sur le système de tâches, seule une simulation permet de conclure à l'ordonnançabilité du système de tâches.

Cependant, lorsqu'on opère la simulation d'un système de tâches, il faut savoir que les ordonnancements dans des contextes de ressources ou de non-préemption ne sont pas réguliers.

Définition I-2-8 : Soit χ un contexte temps réel et A un algorithme d'ordonnancement. On dit que les ordonnancements produits par A sont **réguliers** pour χ si pour tout système de tâches $S=\{\tau_i<r_i,C_i,D_i,P_i>\}_{i=1..n}$ de χ tel que S est fiablement ordonnancé par A, tout système $S'=\{\tau_i'<r_i,C_i'\leq C_i,D_i,P_i>\}_{i=1..n}$ avec pour tout i=1..n, le programme composant τ_i est le même que celui de τ_i' est fiablement ordonnancé par A.

Cette définition est utile car les durées des tâches sont calculées au pire. Dans ce cas, lors de l'exécution des tâches, il est fréquent que la durée effective des tâches soit moindre que la durée utilisée lors de la validation. Il est donc primordial pour un concepteur de se poser la question suivante : *le système que j'ai validé est-il toujours ordonnançable lorsque les durées de mes tâches sont moindres ?*

I-2.1.2.b Approches hors-ligne

On peut penser que les approches en-ligne dynamiques sont plus flexibles que les approches hors-ligne (forcément statiques). En effet, pour les grosses applications pour lesquelles des changements de mode de fonctionnement doivent être envisagés, elles semblent plus adaptées au premier abord. Cependant, la validation de tels cas est tellement délicate lorsque des ressources critiques sont en jeu qu'un travail de complexité équivalente permettrait de produire des séquences correspondant aux différents modes de fonctionnement envisagés.

De plus, bien que les algorithmes hors-ligne soient forcément statiques puisqu'il leur faut connaître toutes les tâches *TRCS*, il est possible de coupler, comme nous le verrons par la suite, un séquence statique avec un algorithme en-ligne dynamique affectant des tâches (non *TRCS*) dans les temps creux laissés par les tâches *TRCS*, ce qui rend l'approche hors-ligne plus flexible.

Enfin, on trouve des systèmes de tâches contraints en terme de ressources critiques pour lesquels aucune approche en-ligne ne peut être validée alors qu'une approche hors-ligne simple peut être mise en œuvre.

I-2.1.2.c Validation par simulation

Nous nous intéresserons au cas par cas aux critères analytiques d'ordonnançabilité des algorithmes en-ligne. Dans ce paragraphe nous montrons l'utilité qu'a une approche par simulation.

Théorème I-2-1 : [BHR 90][LM 80][LW 82] Soit un système de tâches S dans le contexte $\chi=|1|r_i,C_i,D_i,P_i|$, le problème de savoir si S est ordonnançable est co-NP-difficile.

Ce résultat fort, montré par Leung et al., montre que dans le cas où certaines tâches sont différées, même le fait d'avoir à sa disposition un algorithme optimal polynomial pour des tâches indépendantes ne suffit pas à valider le système de tâches en temps polynomial. Il ne peut donc pas y avoir dans ce cas de critère d'analyse nécessaire et suffisant calculable en temps polynomial : les critères d'analyse calculables en temps polynomial sont donc des conditions suffisantes. Il en résulte que des systèmes de tâches fiablement ordonnançables par l'algorithme choisi verront les conditions suffisantes associées non vérifiées. Dans ce cas, il reste à effectuer une simulation.

Théorème I-2-2 : [LM 80] Un système de tâches du contexte $|1|r_i=0,C_i,D_i,P_i|$ est fiablement ordonnancé par un algorithme déterministe[8] A si et seulement si la séquence produite par A entre 0 et $P=ppcm(P_i)$ est valide. $\boldsymbol{P}$ est appelé la **méta-période** du système de tâches, avec *ppcm* le plus petit commun multiple.

Ce théorème montre que la simulation sur une méta-période d'un système de tâches indépendantes simultanées donne une condition nécessaire et suffisante (**CNS**) d'ordonnançabilité. Ce résultat s'étend très simplement aux systèmes de tâches simultanées quelconques.

Théorème I-2-3 : Un système de tâches S du contexte $|p,m|r_i=0,C_i,D_i,P_i$, prmptnoprmpt,prec,resRW| est fiablement ordonnancé par un algorithme déterministe A si et seulement si la séquence produite par A entre 0 et P est valide.

Preuve : A l'instant 0, toutes les tâches sont activées simultanément. Si la séquence produite par A est valide, alors à l'instant P, toutes les tâches doivent être terminées, et sont réactivées.

[8] Un algorithme est déterministe si face à la même configuration de tâches, il prend la même décision

Le système est donc dans le même état à l'instant P qu'à l'instant 0. La séquence est donc infiniment répétée et S est fiablement ordonnancé par A.

A l'inverse, si la séquence produite par A n'est pas valide, alors S n'est pas fiablement ordonnancé par A.

❏

Dans le cas moins trivial où des tâches sont différées, la durée de cette simulation, que nous avons étudiée en détail dans cette thèse, a été étudiée dans [LM 80] dans un cadre réduit.

Théorème I-2-4 : [LM 80] Un système de tâches du contexte $|1|r_i,C_i,D_i,P_i|$ est fiablement ordonnancé par l'algorithme *earliest deadline* (voir section I-2.2.1.b) si et seulement si la séquence produite par *earliest deadline* entre 0 et $max\{r_i\}+2P$ est valide.

Remarque : ce résultat est à notre connaissance le seul résultat portant sur la durée de simulation de systèmes de tâches différées. Dans la section III, nous généralisons ce résultat à toute une classe d'algorithmes d'ordonnancement tout en réduisant la durée de simulation et en l'adaptant au contexte $|1|\ r_i,C_i,D_i,T_i,\ prmptnoprmpt,prec,resRW|$.

I-2.2 Ordonnancement en-ligne

L'ordonnancement en-ligne a été largement étudié dans la littérature, et puisque les algorithmes sont conçus pour être implantés dans le noyau temps réel, les algorithmes en-ligne sont de complexité temporelle faible (le plus souvent linéaire en fonction du nombre de tâches). Ce sont des algorithmes qui sont basés sur une politique d'attribution de priorité : à chaque instant, la tâche non bloquée de plus haute priorité s'exécute. En effet, une tâche peut être bloquée par l'attente de la libération d'une ressource ou l'attente d'un message ou bien d'un événement, et il ne faut pas bloquer le système de tâches pour autant.

A l'origine, la politique d'attribution de priorités utilisée était empirique : le concepteur était chargé d'allouer à la main une priorité à chacune des tâches du système. L'algorithme d'ordonnancement se contentant de calculer, à chaque réveil ou fin de tâche, quelle est la tâche de priorité maximale. Puis des algorithmes de calcul des priorités en fonction des paramètres temporels ont été proposés. Le calcul des priorités se répartit en deux catégories : les algorithmes d'ordonnancement à priorités statiques et ceux à priorités variables.

Les études menées reposent pour la plupart sur des modèles de tâches préemptifs, car le problème de l'ordonnancement de tâches (même indépendantes) non préemptibles est NP-complet [LRKB 77][JSM 91].

Cette partie est donc un état de l'art, loin d'être exhaustif, sur les algorithmes d'ordonnancement dynamiques. Pour trouver des états de l'art plus complets ou spécialisés, le lecteur peut se reporter à [Bla 76b] pour des résultats portant sur la complexité des problèmes d'ordonnancement, [HLR 96] pour une comparaison des affectations de priorités statiques et variables, [XP 93][But 97] pour une approche plus générale des techniques d'ordonnancement, et [SSRB 98] pour une présentation détaillée des résultats relatifs à *earliest deadline*. Les résultats que nous présentons sont donnés en utilisant nos notations.

I-2.2.1 Ordonnancement en-ligne de tâches indépendantes

I-2.2.1.a Algorithmes à priorités statiques

Les algorithmes à priorités statiques sont assez simples à mettre en œuvre une fois que les priorités ont été attribuées, puisque l'ordonnanceur n'a aucune mise à jour des priorités à effectuer. En effet, la priorité de chaque tâche est calculée initialement et ne varie pas au cours du temps.

Dans le cadre des algorithmes à priorités statiques, des conditions analytiques d'ordonnançabilité ont été dégagées dans la littérature. Il faut cependant garder en mémoire que le problème d'étude de faisabilité étant co-NP-difficile lorsque des tâches sont différées, les conditions analytiques ne pourront être nécessaires et suffisantes que pour des cas particuliers tels que ceux où les tâches sont simultanées. Elles sont basées sur la notion d'instant critique.

Définition I-2-9 : Lorsqu'un algorithme à priorités statiques est utilisé, l'**instant critique** d'une tâche est la date d'activation de l'occurrence d'une tâche ayant le pire temps de réponse.

Théorème I-2-5 : [LL 73] Dans le contexte $|1|r_i,C_i,D_i,P_i|$, l'**instant critique** pour toute tâche a lieu lorsqu'elle est réveillée en même temps que toutes les tâches plus prioritaires qu'elle.

L'instant critique a lieu à l'instant *0* lorsque les tâches sont simultanées, mais dans le cas général où certaines tâches sont différées, l'instant critique peut ne jamais avoir lieu si les dates de réveil sont bien choisies par rapport aux périodes. Les conditions nécessaires et suffisantes (ou simplement suffisantes) dégagées dans le contexte $|1|r_i=0,C_i,D_i,P_i|$ sont donc des conditions suffisantes dans le contexte $|1|r_i,C_i,D_i,P_i|$.

Pour un état de l'art détaillé des algorithmes a priorités statiques et des conditions analytiques d'ordonnançabilité, le lecteur peut se référer à [ABDTW 95].

i) Rate Monotonic

L'algorithme le plus représentatif de cette classe d'algorithmes est ***Rate Monotonic***[Ser 72], dénoté ***RM***, qui affecte aux tâches une priorité inversement proportionnelle à leur période. En cas d'égalité parmi les tâches prêtes de plus haute priorité, le choix de la tâche à exécuter est fait arbitrairement parmi les ex-æquo.

exemple I-2-1Soit un système de tâches indépendantes *S* défini temporellement par $S=\{\tau_1<0,2,8,8>,\tau_2<0,2,6,12>,\tau_3<0,1,4,4>\}$. La séquence d'ordonnancement de *S* fournie par *RM* est donnée sur la figure I-2-1. En effet, chaque tâche τ_i se voit allouer une priorité $P_{RM}(\tau_i)$, et d'après les paramètres temporels des tâches de S, l'ordre des priorités est le suivant : $P_{RM}(\tau_3) > P_{RM}(\tau_1) > P_{RM}(\tau_2)$.

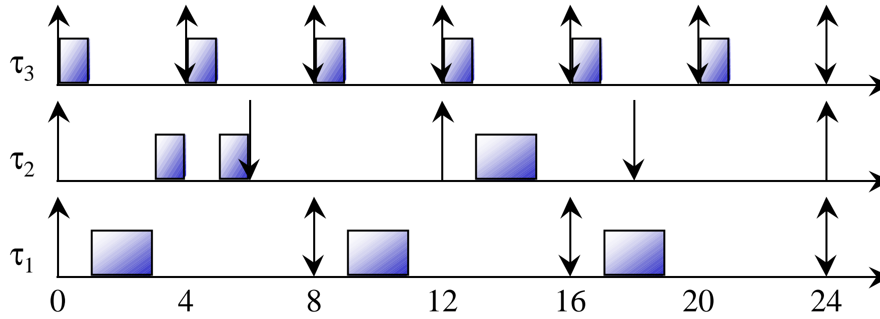


figure I-2-1 : séquence d'ordonnancement produite par Rate Monotonic

Théorème I-2-6 : [LL 73] L'algorithme *RM* est optimal dans le contexte $|1|r_i=0,C_i,D_i=P_i,P_i|$ dans la classe des algorithmes à priorités statiques.

Le résultat d'optimalité de *RM* est basé sur la notion d'instant critique : le pire moment pour une tâche intervient lorsque cette tâche est activée en même temps que les autres tâches de priorité supérieure. C'est en effet pendant cette activation que la tâche sera le plus retardée. Afin de donner une idée de la preuve d'optimalité de *RM*, considérons un système de deux tâches τ_1 et τ_2 telles que $P_1<P_2$. Si l'on décide d'affecter une priorité plus grande à τ_2, lors de l'instant critique de τ_1, τ_2 sera exécutée avant τ_1 et pour que le système soit ordonnançable, il faut que les deux tâches puissent s'exécuter avant l'échéance de τ_1, c'est à dire P_1. La CNS d'ordonnançabilité est donc $C_1+C_2 \leq P_1$. Les tâches τ_1 et τ_2 doivent donc entièrement s'exécuter dans la plus petite période. Par contre, si c'est la tâche τ_1 qui possède la priorité la plus grande, la tâche τ_2 peut s'exécuter sur toute sa période qui est plus grande que celle de τ_1.

Cette politique d'ordonnancement est d'autant plus intéressante qu'il est possible de décider de façon analytique si un système est ordonnançable par *RM* lorsque les tâches sont prises dans le contexte $|1|r_i=0,C_i,D_i=P_i,P_i|$.

En effet, la CS d'ordonnançabilité d'un système de *n* tâches indépendantes à échéance sur requête par *RM* est une borne sur la charge *U* du système de tâches :

Théorème I-2-7 : [LL 73] Soit un système *S* composé de *n* tâches dans le contexte $|1|r_i, C_i, D_i=P_i, P_i|$, si $U_S=\sum_{i=1}^{n}\frac{C_i}{P_i} \leq n\left(2^{1/n}-1\right)$, ce qui tend vers *ln 2* (≈*0.69*) lorsque *n* tend vers l'infini, alors *S* est ordonnançable par *RM*.

Ce test se base sur l'instant critique des tâches, le test se base donc sur le pire cas, c'est à dire sur le cas où $r_i=0$.

La charge de l'exemple de la figure I-2-1 est $U_S=2/8+2/12+1/4= 0.75$, ce qui est inférieur à $3\left(2^{1/3}-1\right)$. Si les tâches avaient été à échéance sur requête, il n'aurait donc pas été nécessaire de construire la séquence fournie par *RM* pour savoir que *S* était ordonnançable par *RM*.

Cependant, même lorsque cette CS n'est pas respectée pour un système *S* donné, la construction de la séquence est utile, car la borne supérieure à la charge des tâches a été fixée expérimentalement aux alentours de *88%* [LSD 89].

Le test précédent étant une CS d'ordonnançabilité, [LSD 89] propose une CNS d'ordonnançabilité dans le cas où les tâches sont simultanées.

Théorème I-2-8 : [LSD 89] Soit un système S composé de n tâches dans le contexte $|1|r_i=0, C_i, D_i=P_i, P_i|$ avec $P_1 \leq \ldots \leq P_n$. S est ordonnançable par *RM* si et seulement si $\forall i$, $1 \leq i \leq n$, $\min_{t \in S_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{P_j} \right\rceil \leq 1$ avec $S_i = \left\{ kP_j, 1 \leq j \leq i, k = 1, \ldots, \left\lfloor \frac{P_i}{P_j} \right\rfloor \right\}$.

L'instant étudié est toujours l'instant critique, c'est à dire l'instant où une tâche est réveillée en même temps que toutes les tâches prioritaires. Dans le cas de tâches simultanées, cet instant est *0*. Cependant l'étude est plus fine que celle menée dans [LL 73], mais aussi plus coûteuse en temps puisque sa complexité est pseudo-polynomiale, de l'ordre de $O(n^2.max_{i=1..n}\{P_i\})$, contre une complexité linéaire pour le test précédent.

L'étude est donc menée sur la première période de chaque tâche τ_i : si à un moment donné, entre l'instant *0* et le réveil d'une tâche plus prioritaire (ou le re-réveil de τ_i), la charge induite par τ_i et les tâches de priorités supérieures est inférieure ou égale à 1, alors τ_i est traitée et respecte son échéance dans sa zone critique. Elle respectera donc toutes ses échéances. En fait, cette étude revient à simuler le système de tâches successivement pendant la première période de chaque tâche.

Un algorithme se basant sur le même principe a été proposé dans [SS 93], de plus il peut s'adapter au cas où l'architecture sous-jacente est répartie.

La littérature a été prolixe en ce qui concerne cette politique, et le livre traitant de la *RMA*[9] [KRPOH 94] reste un ouvrage de référence pour les gens qui ont à implémenter des systèmes temps réel.

Remarquons que l'attribution des priorités par *RM* peut être réprésentée par la signature : $P_{RM} : périodes \rightarrow \mathbb{N}$, où *périodes* est l'ensemble des périodes des tâches, en fait $\mathbb{N}$, étant donné que les priorités ne sont calculées que sur les périodes. Cela implique une implémentation simple du calcul des priorités *RM*.

Existence de CN ou CNS	Contexte d'optimalité
CNS $\|1\|r_i=0,C_i,D_i=P_i,P_i\|$ [LL 73] CS $\|1\|r_i,C_i,D_i,P_i\|$ [LL 73]	$\|1\|r_i=0,C_i,D_i=P_i,P_i\|$ [LL 73] pour une affectation statique des priorités

Tableau I-2-1 : récapitulatif des résultats portant sur RM

ii) Deadline Monotonic

RM a été proposé pour les systèmes de tâches à échéance sur requête. En se reportant à la figure I-2-1, il est clair que si le délai critique de la tâche τ_2 était réduit de *6* à *5* unités de temps, le système ne serait pas ordonnançable par *RM*, alors qu'il aurait été ordonnançable si la tâche τ_2 avait une priorité plus grande que τ_1.

C'est en considérant que la priorité la plus grande devait être assignée à la tâche la plus urgente que ***Deadline Monotonic***[LW 82], aussi appelé ***Inverse Deadline***, noté ***DM***, a été conçu. Cet algorithme, à priorités statiques, affecte une priorité inversement proportionnelle au délai critique. Le lecteur peut se référer à [ABRW 92] pour une démarche générale de validation temporelle de systèmes de tâches par *DM*.

[9] *RMA* : *Rate Monotonic Analysis*

En ce qui concerne le système dont l'ordonnancement est donné en figure I-2-1, les priorités seraient affectées par *DM* de façon à ce que l'ordre suivant soit respecté : $P_{DM}(\tau_3) > P_{DM}(\tau_2) > P_{DM}(\tau_1)$, et la séquence d'ordonnancement produite par *DM* est donnée sur la figure I-2-2.

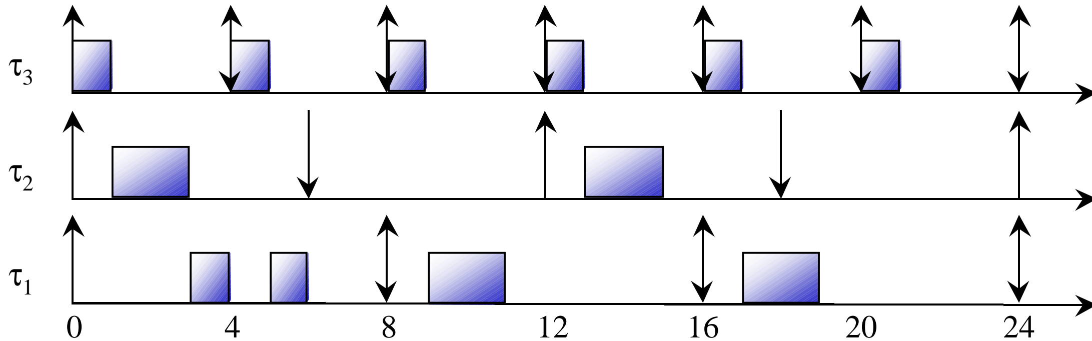


figure I-2-2 : séquence d'ordonnancement produite par Deadline Monotonic

> Théorème I-2-9 : [LW 82] L'algorithme *DM* est optimal dans le contexte $|1|r_i=0,C_i,D_i,P_i|$ dans la classe des algorithmes a priorités statiques.

Il convient de remarquer que le théorème d'optimalité de *DM* n'est valable que sous l'hypothèse $D_i \leq P_i$, qui est faite dans cette thèse. En effet, la non optimalité de *DM* a été montrée dans le contexte $|1|r_i=0,C_i,D_i>P_i,P_i|$ dans [Leh 90]

La signature de l'assignation des priorités par *DM* peut être donnée sous la forme P_{DM} : *délais critiques* $\rightarrow \mathbb{N}$. Comme celles de *RM*, les priorités de *DM* sont simples à calculer.

Des conditions suffisantes d'ordonnançabilité de systèmes de tâches par *DM* peuvent être dégagées :

> Théorème I-2-10 : Soit un système *S* composé de *n* tâches dans le contexte $|1|r_i=0, C_i, D_i, P_i|$, si $\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n\left(2^{1/n}-1\right)$, alors *S* est ordonnançable par *DM*.

Ce test, basé sur celui de [LL 73], est très pessimiste, il peut donc être avantageusement remplacé par une adaptation de la CNS proposée pour *RM*.

> Théorème I-2-11 : [LSST 91] Soit un système *S* composé de *n* tâches dans le contexte $|1|r_i=0, C_i, D_i, P_i|$ avec $D_1 \leq \ldots \leq D_n$. *S* est ordonnançable par *DM* si et seulement si $\forall i$, $1 \leq i \leq n$, $\min_{t \in S_i} \sum_{j=1}^{i} \frac{C_j}{t}\left\lceil \frac{t}{P_j} \right\rceil \leq 1$ avec $S_i = \{D_i\} \cup \left\{ kP_j, 1 \leq j \leq i, k = 1,\ldots,\left\lfloor \frac{D_i}{P_j} \right\rfloor \right\}$.

Cette CNS d'ordonnançabilité est l'adaptation au cas non échéance sur requête du test de [LSD 89]. Elle traduit la CNS suivante : en se basant sur le principe de l'instant critique, le pire instant pour une tâche τ_i (lorsque les tâches sont simultanées) est sa première période : à

l'instant *0*, elle est réveillée en même temps que toutes les tâches plus prioritaires qu'elle. Le principe de cette analyse repose sur l'étude de la charge induite par les réveils successifs de τ_i et des tâches plus prioritaires qu'elle. Si à un moment avant D_i, cette charge se situe en dessous de 1, alors τ_i sera terminée en respectant son échéance, même avec son pire temps de réponse, puisque c'est l'instant critique qui est étudié.

Cette analyse, équivalente à la simulation de l'ordonnancement *DM* pendant la plus grande des périodes des tâches, nécessite un temps pseudo polynomial en $O(n^2.max_{i=1..n}\{P_i\})$.

Dans le cas de *DM*, [SS 93] a proposé un test d'ordonnançabilité de complexité équivalente au test précédent. Dans [MA 98], arguant du fait que les périodes peuvent être arbitrairement longues, un algorithme de test d'ordonnançabilité en $O(n.2^n)$ est proposé.

Existence de CN ou CNS	Contexte d'optimalité
CNS $\|1\|r_i=0,C_i,D_i,P_i\|$ [LSST 91] [MA 98] [SS 93] [Har 87] [Jos 85] [JP 86] [ABRW 91]	$\|1\|r_i=0,C_i,D_i,P_i\|$ [LW 82] pour une affectation statique des priorités

Tableau I-2-2 : récapitulatif des résultats relatifs à DM

iii) Affectation de priorités par l'algorithme d'Audsley

Bien qu'optimal dans le contexte $|1|r_i=0, C_i, D_i, P_i|$, Audsley a montré dans [Aud 91] que *DM* ne l'était pas dans le contexte $|1|r_i, C_i, D_i, P_i|$. Audsley a exhibé un algorithme d'affectation optimale de priorités statiques dans ce contexte, qui l'est aussi d'ailleurs dans le contexte $|1|r_i, C_i, D_i>P_i, P_i|$.

Soit *S* un système de *n* tâches. L'algorithme affecte une priorité de {0,…,n−1} à chaque tâche du système de façon distincte. Pour cela, il commence par choisir une tâche pouvant être ordonnancée avec la plus petite priorité $n-1$ tout en respectant ses contraintes temporelles et lui affecte la priorité $n-1$. Au $k^{ième}$ pas de l'algorithme, on choisit parmi les tâches non affectées une tâche respectant ses contraintes temporelles avec la plus petite priorité non affectée, en l'occurrence $n-k$. [Aud 91] montre qu'à chaque pas de l'algorithme, si plusieurs tâches peuvent se voir affecter la plus faible priorité restante, alors le choix de l'une d'entre elles peut être fait arbitrairement sans nuire à l'ordonnançabilité future du système. Si les *n* pas nécessaires à l'affectation des *n* priorités sont effectués avec succès, le système est ordonnançable, dans le cas contraire, aucune affectation statique de priorités ne peut ordonnancer le système.En tout, l'algorithme prend en compte au plus $O(n^2)$ affectations de priorités, chacune étant testée à l'aide d'un test d'ordonnançabilité, qui reste co-NP-difficile dans le contexte considéré par Audsley.

Théorème I-2-12 : [Aud 91] L'affectation de priorités d'**Audsley** est optimale dans le contexte $|1|r_i, C_i, D_i, P_i|$ pour les algorithmes à priorités statiques.

Existence de CN ou CNS	Contexte d'optimalité
CNS $\|1\|r_i,C_i,D_i,P_i\|$ [Aud 91] automatiquement vérifiée lors de l'affectation (en temps exponentiel)	$\|1\|r_i,C_i,D_i,P_i\|$ [Aud 91] pour une affectation statique des priorités

Tableau I-2-3 : récapitulatif des résultats relatifs à l'affectation de priorités d'Audsley

iv) Analyse générale des affectations statiques des priorités

Des conditions analytiques d'ordonnançabilité ont été étudiées dans le cas général des algorithmes à priorités statiques. Toujours basées sur l'instant critique des tâches, elles ne sont nécessaires et suffisantes que lorsque les tâches sont simultanées.

[Har 87][Jos 85][JP 86][ABRW 91] ont étudié des algorithmes de dilatation du temps basés sur le Théorème I-2-13, et choisis afin d'optimiser le nombre de points dans le temps étudiés.

> Théorème I-2-13 : [Har 87][Jos 85][JP 86] Soit un système de tâches S dans le contexte $|1|r_i=0,C_i,R_i,P_i|$ et une affectation de priorités telle que $priorité(\tau_1)>priorité(\tau_2)>\ldots>priorité(\tau_n)$. Si l'équation $R=C_i+\sum_{j=1}^{i}\left\lceil \frac{R}{P_j} \right\rceil C_j$ a une solution $R\in[0, D_i]$, alors la tâche τ_i respecte ses échéances, de plus, son pire temps de réponse est donné par la plus petite solution (positive) de cette équation.

I-2.2.1.b Algorithmes à priorités variables

A cause des limites de charge processeur théorique imposées aux algorithmes à priorité statique, des algorithmes d'ordonnancement à priorités variables ont été proposés. Ces algorithmes diffèrent des algorithmes précédemment étudiés puisqu'ils affectent aux tâches des priorités variant au cours de la vie de l'application.

Les détracteurs de ces algorithmes leur imputent une grande surcharge processeur, ou *overhead* en anglais. Les priorités doivent en effet être recalculées au moins à chaque réveil de tâche, mais ce calcul est linéaire en temps en fonction du nombre de tâches.

Nous nous attacherons à étudier les contextes d'optimalité des algorithmes présentés, ainsi que les CNS et CS établies dans la littérature. Comme pour les algorithmes à priorités statiques, l'étude d'ordonnançabilité des systèmes de tâches différées est plus délicate que celle des tâches simultanées.

> Théorème I-2-14 : [RCM 96] Soit S un système de tâches pris dans le contexte $|1|r_i=0,C_i,D_i,P_i|$, si S est ordonnançable, alors tout système S' ayant les mêmes caractéristiques que S sauf que les r_i peuvent être décalés (i.e. certaines tâches sont différées) est ordonnançable.

Ce théorème nous permet de considérer que toute CNS ou CS dans le contexte $|1|r_i=0,C_i,D_i,P_i|$ pour un algorithme optimal dans le contexte $|1|r_i,C_i,D_i,P_i|$ est une CS dans le contexte $|1|r_i,C_i,D_i,P_i|$. Nous verrons que c'est le cas des deux algorithmes présentés.

i) Earliest Deadline

Le principe d'***Earliest Deadline*** *ED* est d'affecter la plus grande priorité à la tâche dont l'échéance est la plus proche. Historiquement, cet algorithme a été proposé dans un contexte temps réel dans [Ser 72] (où il était nommé ***Relative Urgency***) [LL 73] (dans lequel il était nommé ***Deadline Driven Scheduling***)et est inspiré des travaux en ordonnancement *job shop* de [Jac 55][McN 59] repris dans [CMM 67].

Théorème I-2-15 : [Jac 55] La latence maximale et le retard maximal imposés à des tâches (dans un contexte $|1|r_i=0,C_i,D_i|$) sont minimisés par l'ordonnancement des tâches dans l'ordre des échéances croissantes.

La fonction P_{ED} d'attribution des priorités par *ED* varie donc au cours du temps, et la signature de cette fonction est P_{ED} : *dates de réveil*×*délais critiques*×*périodes*×*temps* → $\mathbb{N}$, avec *dates de réveil*⊆ $\mathbb{N}$, *délais critiques*⊆ $\mathbb{N}$ et *périodes*⊆ $\mathbb{N}$. D'une part, la priorité *ED* est donc calculée avec plus de paramètres, mais en plus, cette priorité varie au cours du temps. La complexité relative d'implémentation d'un noyau temps réel basé sur *ED* par rapport à un algorithme à priorités statiques explique en grande partie la rareté des noyaux temps réel dont la politique d'ordonnancement est basée sur *ED*.

Pourtant, *ED* est l'algorithme d'ordonnancement en ligne le plus puissant à ce jour, puisqu'il est optimal pour les systèmes de tâches indépendantes.

Théorème I-2-16 : [Der 74][Lab 74] *Earliest Deadline* est optimal dans le contexte $|1|r_i,C_i,D_i,P_i|$.

Dans le contexte $\chi=|1|r_i,C_i,D_i,P_i|$, on a donc $RM<_\chi DM<_\chi \text{Audsley}<_\chi ED$.

Soit *S* tel que $S=\{<0,2,8,8>,<0,4,5,12>,<0,1,4,4>\}$. Sa charge est $U_S=2/8+4/12+1/4=11/12\approx0.92$. Le système ainsi obtenu n'est pas ordonnançable par *DM* (voir figure I-2-3), il n'est donc ordonnançable par aucun algorithme a priorités statiques, puisque nous nous trouvons dans le contexte $|1|r_i=0,C_i,D_i,P_i|$. Voyons sur la figure I-2-4 la séquence produite par ED.

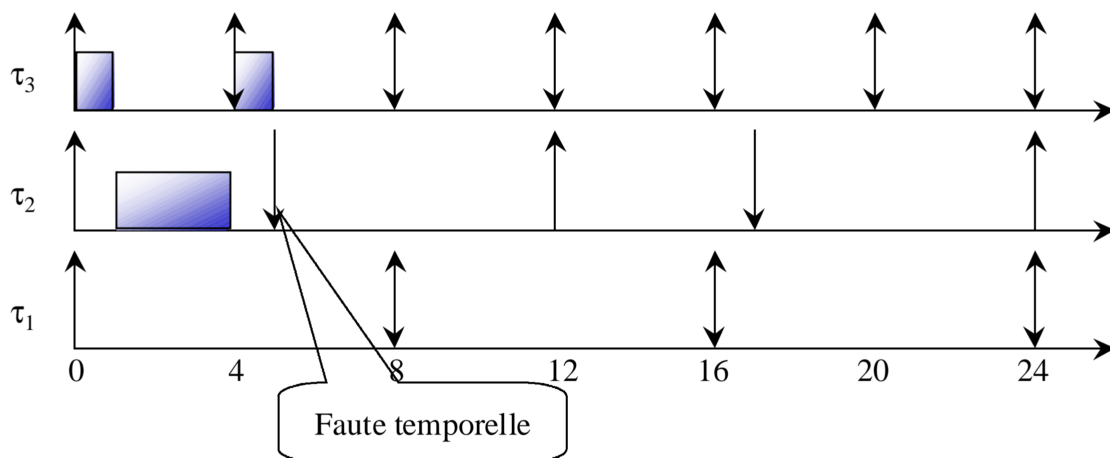


figure I-2-3 : séquence d'ordonnancement produite par Deadline Monotonic

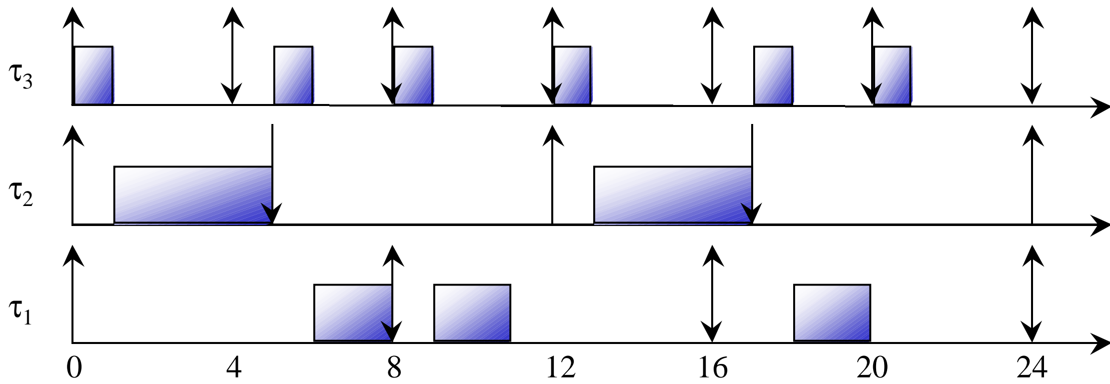


figure I-2-4 : séquence d'ordonnancement produite par Earliest Deadline

Comme pour les algorithmes à priorités statiques, il est primordial de dégager des conditions analytiques pour lesquelles on est certain qu'un système donné est ordonnançable par *ED*.

Théorème I-2-17 : [LL 73] Un système de tâches *S* est ordonnançable dans le contexte $|1|r_i=0,C_i,D_i=P_i,P_i|$ si et seulement si $U=\sum_{i=1}^{n}\frac{C_i}{P_i}\leq 1$.

Dans le cas où les tâches sont simultanées et à échéance sur requête, *ED* permet d'obtenir une borne au taux d'utilisation processeur qui est de *100%*, c'est donc un algorithme d'ordonnancement optimal dans ce cas particulier puisque $U\leq 1$ est une condition nécessaire d'ordonnançabilité pour tout algorithme d'ordonnancement. Ce théorème a été étendu dans le contexte $|1|r_i=0,C_i,D_i\geq P_i,P_i|$ dans [BRM 90].

Cependant, dans le cas contraire, l'ordonnançabilité d'un système de tâches par *ED* est plus difficile. L'utilisation du théorème portant sur la durée de simulation permet l'obtention d'une CNS d'ordonnançabilité par *ED*. Précisons qu'à l'origine, Leung et Merrill l'avaient énoncé en deux points :

Théorème I-2-18 : [LM 80][LW 82] Un système de tâches *S* est ordonnançable par *ED* dans le contexte $|1|r_i,C_i,D_i,P_i|$ si et seulement si : (1) l'ordonnancement produit par simulation sur l'intervalle $[0, max_{i=1..n}\{r_i\}+2P]$ est valide, (2) l'état du système est identique aux dates $max_{i=1..n}\{r_i\}+2P$ et $max_{i=1..n}\{r_i\}+P$.

Le second point du théorème a été montré inutile dans [BHR 90] puisqu'il est toujours satisfait lorsque $U\leq 1$ et qu'*ED* est utilisé. Baruah et al. ont réduit cette durée de simulation exponentielle à une durée pseudo-polynomiale dans le cas de tâches simultanées non à échéance sur requête.

Théorème I-2-19 : [BHR 90] Un système de tâches S, de charge U<1, est ordonnançable par *ED* dans le contexte $|1|r_i=0,C_i,D_i,P_i|$ si et seulement si toutes les échéances sont respectées dans l'intervalle $\left[0..\frac{U}{1-U}\max\{P_i - D_i\}\right]$.

Cette borne a été améliorée dans [RCM 96], et [ZS 94]. Cependant, à notre connaissance, la seule CNS d'ordonnançabilité d'un système de tâches dans le contexte $|1|r_i,C_i,D_i,P_i|$ est la simulation sur $[0..max_{i=1..n}\{r_i\}+2P]$, ce qui est un temps exponentiel (voir annexe I-1).

Existence de CN ou CNS	Contexte d'optimalité
CNS $\|1\|r_i=0,C_i,D_i=P_i,P_i\|$ [LL 73] CNS $\|1\|r_i=0,C_i,D_i,P_i\|$ [BHR 90] CS $\|1\|r_i,C_i,D_i,P_i\|$ [BHR 90][RCM 96]	$\|1\|r_i,C_i,D_i,P_i\|$ [Der 74] [Lab 74]

Tableau I-2-4 : récapitulatif des résultats relatifs à ED

ii) Least Laxity

Pour compléter l'étude des principaux algorithmes en-ligne d'ordonnancement dans le cadre des systèmes de tâches indépendantes, ce paragraphe propose l'étude de l'algorithme ***Least Laxity***, noté ***LL***. Il est basé sur la laxité des tâches : à chaque instant, la tâche de plus petite laxité a la plus forte priorité.

Cet algorithme est capable d'ordonnancer les mêmes systèmes qu'*ED* puisqu'il est optimal dans le même contexte.

Théorème I-2-20 : [Mok 83][DM 89] *Least Laxity* est optimal dans le contexte $|1|r_i,C_i,D_i,P_i|$.

Donc les algorithmes *ED* et *LL* ont la même puissance d'ordonnançabilité dans le contexte $\chi=|1|r_i,C_i,D_i,P_i|$, on peut donc écrire $ED=_\chi LL$.

Cependant, *LL* impose une surcharge processeur importante, puisque les priorités doivent être recalculées à chaque unité de temps (contre tout réveil de tâche pour *ED*). De plus, comme l'illustre la figure I-2-5, il est très coûteux en changements de contexte. Ceci est évidemment coûteux en temps, bien qu'on ait tendance à négliger le coût des changements de contexte lorsque les quanta de temps sont relativement longs. Pour finir sur les inconvénients de *LL* par rapport à *ED* en environnement monoprocesseur, ajoutons que les temps de réponse moyens sont meilleurs avec *ED* qu'avec *LL*, la figure I-2-5 en est d'ailleurs une illustration : pour cet exemple, en cas d'égalité de priorité, la tâche τ_2 est choisie.

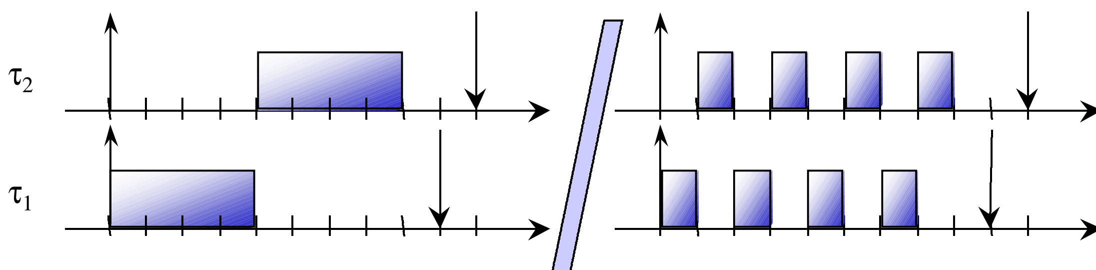


figure I-2-5 : Earliest Deadline / Least Laxity

C'est donc pour des raisons d'implémentation plus coûteuse que celle d'*ED*, sans gain de puissance d'ordonnancement, que *LL* est peu utilisé dans le contexte monoprocesseur.

Existence de CN ou CNS	Contexte d'optimalité
CNS $|1|r_i=0,C_i,D_i=P_i,P_i|$ [LL 73] CNS $|1|r_i=0,C_i,D_i,P_i|$ [BHR 90] CS $|1|r_i,C_i,D_i,P_i|$ [BHR 90][RCM 96]	$|1|r_i=0,C_i,D_i,P_i|$ [Mok 83][DM 89]

Tableau I-2-5 : récapitulatif des résultats relatifs à LL

I-2.2.1.c Récapitulatif sur l'ordonnancement de tâches indépendantes en environnement monoprocesseur

Nous reprenons les résultats d'optimalité, les CS et les CNS d'ordonnançabilité présentés précédemment pour le contexte $|1|ri,C_i,D_i,P_i|$ et ses cas particuliers.

Attribution des priorités	Existence de CN ou CNS	Contexte d'optimalité
RM	CNS $|1|r_i=0,C_i,D_i=P_i,P_i|$ [LL 73] CS $|1|r_i,C_i,D_i,P_i|$ [LL 73]	$|1|r_i=0,C_i,D_i=P_i,P_i|$ [LL 73] pour une affectation statique des priorités
DM	CNS $|1|r_i=0,C_i,D_i,P_i|$ [LSST 91] [MA 98] [SS 93] [Har 87] [Jos 85] [JP 86] [ABRW 91]	$|1|r_i=0,C_i,D_i,P_i|$ [LW 82] pour une affectation statique des priorités
Audsley	CNS $|1|r_i,C_i,D_i,P_i|$ [Aud 91] automatiquement vérifiée lors de l'affectation (en temps exponentiel)	$|1|r_i,C_i,D_i,P_i|$ [Aud 91] pour une affectation statique des priorités
ED	CNS $|1|r_i=0,C_i,D_i=P_i,P_i|$ [LL 73] CNS $|1|r_i=0,C_i,D_i,P_i|$ [BHR 90] CS $|1|r_i,C_i,D_i,P_i|$ [BHR 90][RCM 96]	$|1|r_i,C_i,D_i,P_i|$ [Der 74] [Lab 74]
LL	idem que *ED*	$|1|r_i=0,C_i,D_i,P_i|$ [Mok 83][DM 89]

Tableau I-2-6 : récapitulatif des résultats portant sur les algorithmes d'ordonnancement des systèmes de tâches indépendantes

De plus, dans le cadre des tâches indépendantes, les algorithmes d'ordonnancement sont réguliers, la simulation [LM 80] reste une CNS pour tous les cas où on ne peut valider autrement.

I-2.2.2 Ordonnancement en-ligne de tâches communicantes

Le modèle présenté en section précédente est limitatif puisqu'il ne permet pas de représenter des tâches communicantes. Or, la plupart des applications temps réel nécessitent des communications entre les tâches, qui introduisent des contraintes de précédence entre certaines parties des tâches. Les notions de communication et de précédence étant étroitement liées, un terme pourra être utilisé à la place de l'autre et vice versa.

Il semble que seules les communications asynchrones par boîtes aux lettres soient prises en compte dans les différents modèles étudiés. Cela n'est pas limitatif pour les communications synchrones étant donné que les communications synchrones peuvent être obtenues à l'aide de communications asynchrones (§I-1.3.1.b). Les événements, quant à eux, sont très utilisés lors

d'une approche temps réel synchrone mais sont proscrits lors d'une approche temps réel asynchrone. En effet, lors d'une programmation événementielle, aucun graphe de précédence « statique » ne peut être construit.

Plutôt que de modifier les algorithmes d'ordonnancement classiques énoncés plus haut et de refaire tout un travail d'adaptation des CN et CNS d'ordonnançabilité déjà obtenues, la communauté temps réel a pris le parti de découper les tâches communicantes autour des points de communication et de modifier leurs paramètres temporels (en l'occurrence les r_i et les D_i) afin de pouvoir utiliser les algorithmes classiques sans modification. Le principe du découpage des tâches, que nous appelons **découpage en forme normale**, repose sur le principe suivant : les algorithmes classiques (hors affectation d'Audsley) étant basés sur des affectations de priorité au vu des paramètres temporels, il faut s'assurer que la partie d'une tâche précédée par une autre aura une priorité inférieure à la partie de la tâche la précédant.

Rappelons que nous supposons les tâches abstraites à leur durée, ce qui sous-entend que les boucles les composant ont été déroulées, et que les structures alternatives ont été abstraites à leur pire cas. Le découpage en forme normale se passe de la façon suivante : soit une tâche $\tau_i<r_i,C_i,D_i,P_i>$ émettant et recevant des messages, comme cela est représenté sur la figure I-2-6. La tâche τ_i est découpée en sous-tâches autour des points de communication de sorte à ce que les attentes de messages par une sous-tâche soient en début de tâche, et que les émissions de messages soient en fin de tâche. Des contraintes de précédence, symbolisées par le symbole ∅ sur la figure, sont introduites afin de conserver la séquentialité de la tâche τ_i.

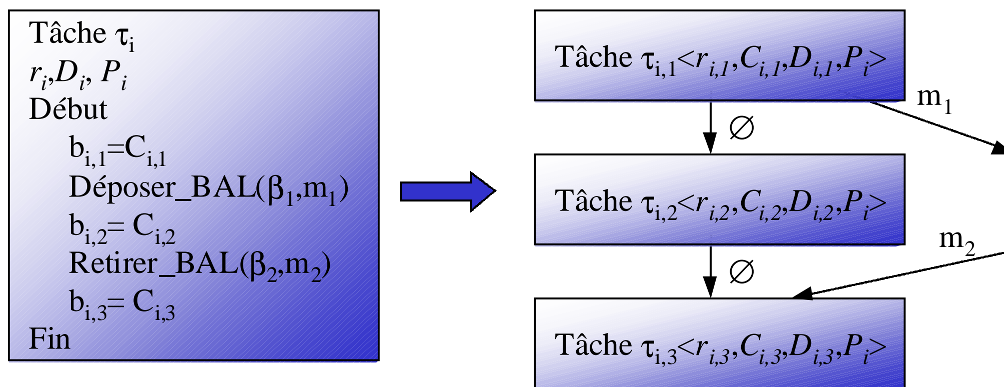


figure I-2-6 : découpage d'une tâche autour des points de communication

Ce découpage permet d'obtenir un ordre partiel sur les tâches représentant les contraintes de précédence des sous-tâches, noté $\tau_i \rightarrow \tau_j$ pour τ_i précède τ_j.

Les précédences sont souvent représentées graphiquement par des arcs orientés. Étant donné que les précédences ne peuvent pas admettre de cycles, les précédences des systèmes de tâches doivent pouvoir être représentées par un graphe orienté sans cycle. Sur la figure I-2-7, la tâche τ_1 précède la tâche τ_2, qui précède la tâche τ_3 et la tâche τ_4.

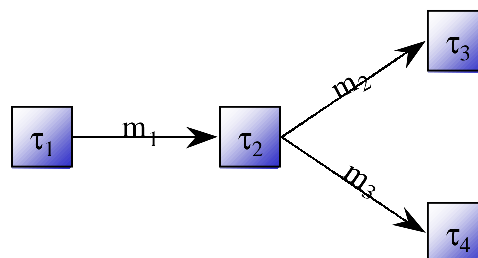


figure I-2-7 : un graphe de précédence d'un système de tâches

On peut déplorer le manque de lisibilité du modèle de tâches, puisque il est nécessaire de connaître la description temporelle des tâches et leur graphe de précédence pour connaître totalement un système de tâches. De plus, la représentation graphique des contraintes de précédence nécessite que les arcs entrants correspondent à une (ou des) instruction d'attente de message en début de tâche et que les arcs sortants correspondent à une (ou des) émission de message en fin de tâche.

Un exemple de découpage en forme normale d'un système de tâches est donné en figure I-2-8.

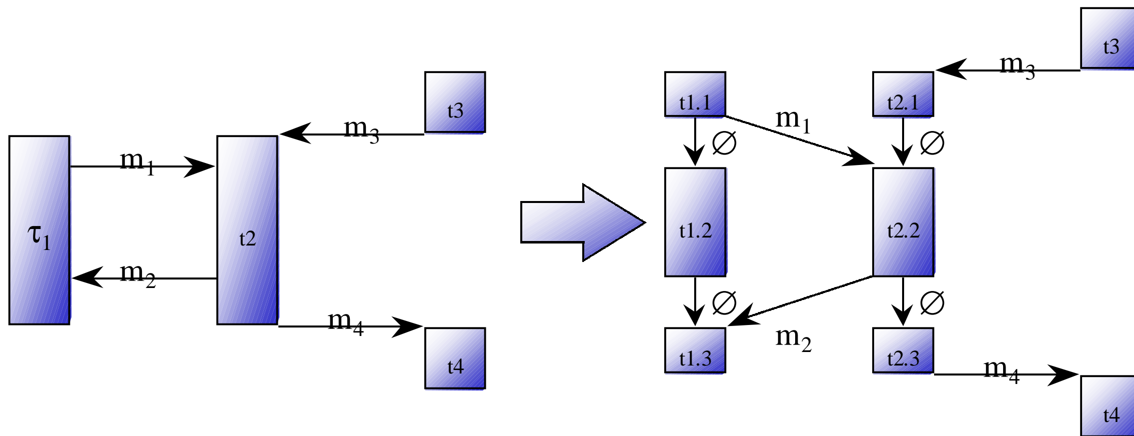


figure I-2-8 : découpage d'un système de tâches en forme normale

Chaque tâche est découpée en sous-tâches dans lesquelles les émissions se trouvent en fin d'exécution et les réceptions au début. Ainsi, les contraintes de précédence portent entre tâches et non plus sur des parties de tâches.

Blazewicz [Bla 76a] a montré comment modifier les paramètres date de réveil et délai critique des sous-tâches afin que les contraintes de précédence soient implicitement respectées par l'algorithme *ED*.

Théorème I-2-21 : [Bla 76a] *ED* est optimal dans le contexte $|1|r_i,C_i,D_i,P_i,précformenormale|$ si après l'éclatement en forme normale d'une tâche τ_i en sous-tâches $\tau_{i,1},\ldots,\tau_{i,k}$, les paramètres des sous-tâches sont choisis de la façons suivante :

- $D_{i,j} = min\{D_i, min_{\tau_{i,j}\rightarrow\tau_{q,r}}\{D_{q,r}-C_q\}\}$ à partir des tâches sans successeurs et en remontant à chaque pas vers les tâches dont tous les successeurs ont été traités.
- $r_{i,j} = max\{r_i, max_{\tau_{q,r}\rightarrow\tau_{i,j}}\{r_{q,r}+C_q\}\}$ à partir des tâches sans prédécesseurs et en descendant à chaque pas vers les tâches dont tous les prédécesseurs ont été traités.

Rappelons que par hypothèse, les tâches communicantes ont la même période, ce qui implique que les modifications opérées sur les sous-tâches se répercutent directement sur chaque occurrence de celles-ci. Ce théorème est un outil de transformation des sous-tâches issues de la mise en forme normale d'un système afin que l'algorithme *ED* considère le système comme constitué de tâches indépendantes, tout en respectant les contraintes de précédence de façon implicite.

Il est à noter que ce théorème ne fonctionne que si les tâches communicantes ont même période. Cette contrainte est levée à un coût exponentiel dans [RC 99], afin de prendre en compte des contraintes de précédence généralisées : une tâche émettrice peut émettre au cours de chacune de ses occurrences plusieurs messages vers une réceptrice, qui, elle, peut en lire un nombre différent, le tout à condition que les fréquences d'émission et de réception de messages soient les mêmes. En effet, si les émissions sont moins fréquentes que les réceptions (qui sont bloquantes), la tâche réceptrice associée ne pourra pas respecter ses échéances indéfiniment. De façon similaire, si c'est la fréquence d'émission d'un message qui est plus grande que la fréquence de réception, et que la boîte aux lettres est sans écrasement, alors c'est la tâche émettrice qui ne respectera pas une échéance. Si la boîte aux lettres est avec écrasement, alors au bout d'un certain nombre de périodes, la boîte aux lettres étant toujours pleine, il n'y a plus lieu de parler de précédences. Dans ce cas, la boîte aux lettres peut être considérée comme une zone mémoire commune protégée par sémaphore.

Le même ajustement des paramètres temporels des tâches est effectué lorsque des algorithmes à priorités statiques sont utilisés. Dans ce cas les contraintes sont qu'une tâche émettrice ait une priorité plus forte que les tâches réceptrices associées, et que sa première activation ait lieu avant ou en même temps que les premières activations des réceptrices : cette avance est conservée tout le long de la vie de l'application puisqu'il est imposé que les tâches communicantes aient même période.

Théorème I-2-22 : Soit $S=\{\tau_i<r_i,C_i,D_i,P_i>\}_{i=1..n}$ une configuration de tâches en forme normale (i.e. les attentes de messages ne peuvent être qu'en début de tâche et les envois en fin de tâche) dans le contexte $|1|r_i,C_i,D_i,P_i,precformenormale|$, et soit $\rightarrow$ un ordre partiel entre les tâches : $\tau_i\rightarrow\tau_j$ signifie que chaque occurrence de τ_i doit être entièrement exécutée avant que l'occurrence correspondante de τ_j ne débute.

S est ordonnançable avec respect de l'ordre partiel $\rightarrow$ par RM si et seulement si $S^*=\{\tau_i^*<r_i^*,C_i,D_i^*,P_i>\}_{i=1..n}$ est ordonnançable dans le contexte $|1|r_i,C_i,D_i,P_i|$ avec :

(1) $r_i^*=max\{r_i,max_{\tau_j\rightarrow\tau_i}\{r_j^*\}\}$ en traitant d'abord les tâches sans prédécesseurs et en descendant à chaque pas vers les tâches dont tous les prédécesseurs ont été traités.

(2) $D_i^*=D_i-(r_i^*-r_i)$ afin de respecter les mêmes échéances absolues.

(3) Si deux tâches ont même période avec $\tau_i\rightarrow\tau_j$, alors $P_{RM}(\tau_i)>P_{RM}(\tau_j)$.

Preuve : la preuve, dont on peut trouver une version dans [RC 99], se base sur le fait que le premier point permet de faire en sorte qu'une tâche précédant une autre ne puisse être activée après celle-ci, le second point permet de refléter les échéances absolues par rapport aux modifications de dates de réveil, enfin le troisième point permet de faire en sorte qu'une tâche précédant une autre tâche soit traitée naturellement par l'ordonnanceur avant cette dernière.

Le même ajustement des paramètres peut être effectué pour l'algorithme d'Audsley. Pour l'algorithme *DM*, les modifications sont expliquées ci-dessous.

Théorème I-2-23 : Soit $S=\{\tau_i<r_i,C_i,D_i,P_i>\}_{i=1..n}$ une configuration de tâches en forme normale dans le contexte $|1|r_i,C_i,D_i,P_i,precformenormale|$, et soit $\rightarrow$ un ordre partiel entre les tâches.

S est ordonnançable avec respect de l'ordre partiel $\rightarrow$ par *DM* si et seulement si $S^*=\{\tau_i^*<r_i^*,C_i,D_i^*,P_i>\}_{i=1..n}$ est ordonnançable dans le contexte $|1|r_i,C_i,D_i,P_i|$ avec :

(1) $r_i^*=max\{r_i,max_{\tau_j\rightarrow\tau_i}\{r_j^*\}\}$ en traitant d'abord les tâches sans prédécesseurs et en descendant à chaque pas vers les tâches dont tous les prédécesseurs ont été traités.

(2) $D_i^*=min\{D_i-(r_i^*-r_i), min_{\tau_i\rightarrow\tau_j}\{D_j^*\}\}$ afin de respecter les mêmes échéances absolues.

(3) Si deux tâches ont même délai critique avec $\tau_i\rightarrow\tau_j$, alors $P_{DM}(\tau_i)>P_{DM}(\tau_j)$.

Dans le cadre des algorithmes à priorités, il est donc relativement simple d'intégrer les contraintes de précédence dans les sous-tâches après un découpage en forme normale pourvu que les périodes des tâches soient identiques (voir [RC 99] pour lever cette limitation).

Des travaux ont porté notamment dans [CSB 90] sur la modification en ligne des paramètres temporels des tâches afin d'accepter des tâches apériodiques et de refléter les contraintes de précédence.

I-2.2.3 Partage de ressources

Il est bien rare dans une application temps réel de pouvoir se passer de l'utilisation de ressources critiques, ne serait-ce que pour protéger l'accès à une zone mémoire partagée. Il était donc indispensable d'intégrer les partages de ressources aux algorithmes d'ordonnancement. Cependant, le problème de l'ordonnancement de tâches partageant des ressources est NP-difficile.

Théorème I-2-24 : [Mok 83] Le problème de décider si un système de tâches est ordonnançable dans le contexte $|1|r_i,C_i,D_i,P_i,res|$ est NP-difficile.

La preuve de ce théorème a été obtenue par réduction au problème 3-partitions. Bien que vérifier l'ordonnançabilité d'un système de tâches dans le contexte $|1|r_i,C_i,D_i,P_i|$ soit co-NP-difficile, nous avons vu qu'il existait dans ce contexte des algorithmes en-ligne optimaux. Cela n'est pas le cas dès lors que des ressources sont en jeu.

Théorème I-2-25 : [Mok 83] Dans le contexte $|1|r_i,C_i,D_i,P_i,res|$, aucun algorithme en-ligne n'est optimal.

En dehors du phénomène connu d'interblocage, l'un des phénomènes les plus caractéristiques engendré par l'adjonction de ressources critiques à un système est l'inversion de priorité. Une tâche peu prioritaire détenant une ressource s'exécute, elle est interrompue pour laisser le processeur à une tâche de priorité moyenne, et lorsqu'une tâche de haute priorité, nécessitant la ressource de la tâche de basse priorité, est activée, celle-ci doit attendre la terminaison de la tâche de priorité intermédiaire, puis la libération de la ressource par la tâche de basse priorité, avant de pouvoir s'exécuter. Dans ce cas, le temps perdu par la tâche de haute priorité à cause de l'inversion de priorité n'est pas borné, car d'autres tâches de priorité intermédiaire peuvent empêcher la tâche de basse priorité de s'exécuter. Un exemple d'inversion de priorité est donné sur la figure I-2-9, où l'ordre des priorités est $Prio(\tau_1)>Prio(\tau_2)>Prio(\tau_3)$.

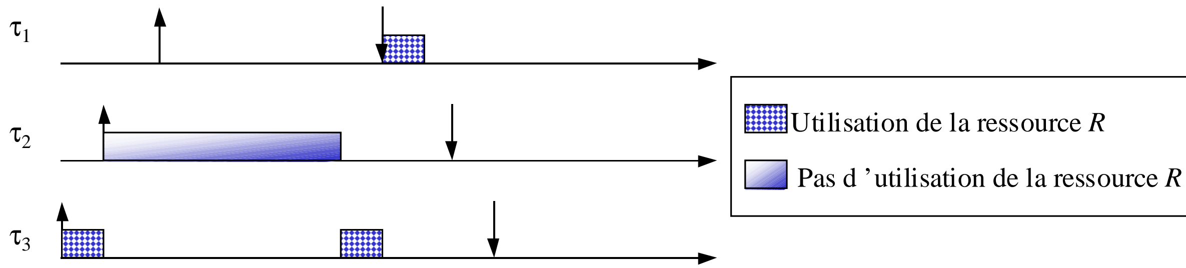


figure I-2-9 : la tâche τ_1 est victime d'une inversion de priorité de la part de τ_2 à cause de τ_3

Afin de pouvoir expliquer les problèmes de blocage qui se posent, il convient de définir un vocabulaire adéquat :

Définition I-2-10 : Il existe différentes formes de blocage, la plus naturelle étant le **blocage direct** : une tâche τ_i est **bloquée** par τ_j si celle-ci est de priorité native moindre que τ_i et détient une ressource nécessaire à l'avancement de τ_i.

Soulignons qu'une tâche en attente d'une ressource détenue par une tâche de priorité supérieure n'est pas considérée comme bloquée. Sur la figure I-2-9, τ_1 est bloquée par τ_2, et subit donc une inversion de priorité de la part de τ_2.

Pour pallier ce problème, les algorithmes classiques d'ordonnancement ont été enrichis de protocoles de gestion de ressources, tels le protocole à priorité héritée, le protocole à priorité plafond ou le protocole de gestion des ressources à piles. La propriété de tels protocoles est d'éliminer les inversions de priorité et de borner les durées de blocage. Ces protocoles sont brièvement présentés dans les paragraphes suivants.

Les **interblocages**[10] peuvent être évités de façon classique, par exemple si les ressources sont prises dans un ordre prédéfini, ou bien naturellement par l'emploi de certains protocoles de gestion de ressources.

Dans cette partie, lorsqu'un système de tâches utilise des ressources R_1, R_2, ..., on décrit les tâches sous la forme τ_i=<r_i,C_i,D_i,P_i,{<α_1, β_1, γ_1>,< α_2, β_2, γ_2>,...}> où pour chaque ressource R_j, le triplet <α_j, β_j, γ_j> signifie <temps d'exécution avant prise de la ressource R_j, pendant, après>. Cette notation n'autorise pas de prises/libérations multiples d'une même ressource par une tâche, mais c'est la notation classiquement utilisée dans l'ordonnancement en-ligne. De plus, elle suppose que chaque ressource n'a qu'une instance.

Il est important de remarquer que contrairement au cas où les tâches sont indépendantes, l'utilisation de ressources critiques empêche de considérer la durée au pire des tâches comme étant un pire cas.

Théorème I-2-26 : Les systèmes de tâches dans le contexte $|1|r_i,C_i,D_i,P_i,res|$ ne sont pas réguliers lorsque des algorithmes conservatifs sont utilisés.

Preuve : La preuve de ce théorème est donnée sur un exemple, exhibant un cas où un système de tâches est ordonnançable par un algorithme conservatif (au plus tôt) en considérant les pires durées des tâches, et ne l'est plus lorsque la durée d'une tâche est réduite.

En effet, considérons le système *S* contenant une ressource tel que :

[10] Un interblocage a lieu lorsque toutes les tâches sont bloquées en attente de ressources détenues par d'autres tâches, elles-aussi bloquées. Il faut au minimum deux ressources pour donner naissance à un interblocage.

S={<0,6,16,16,{<0,6,0>}>,<0,2,6,8{<0,2,0>}>,<0,6,15,16{<6,0,0>}>}.

La séquence d’ordonnancement produite par *ED* qui est un algorithme conservatif est valide (voir figure I-2-10).

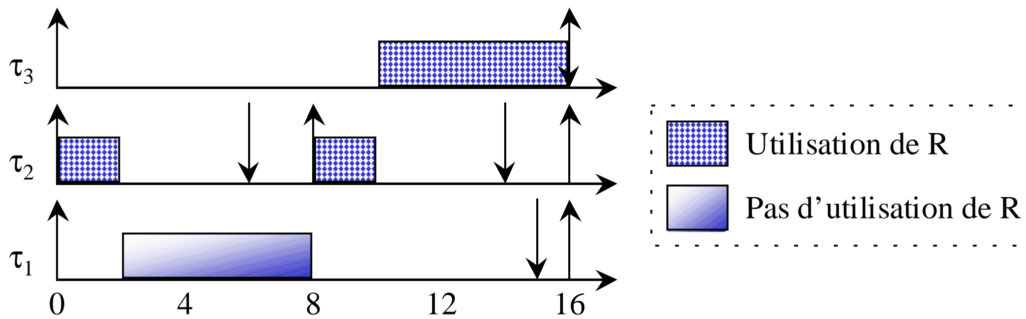


figure I-2-10 : séquence d’ordonnancement valide d’un système partageant une ressource

Cependant, la figure I-2-11 si la durée réelle de τ_1 n’est pas 6 mais 5, alors le système n’est plus ordonnançable.

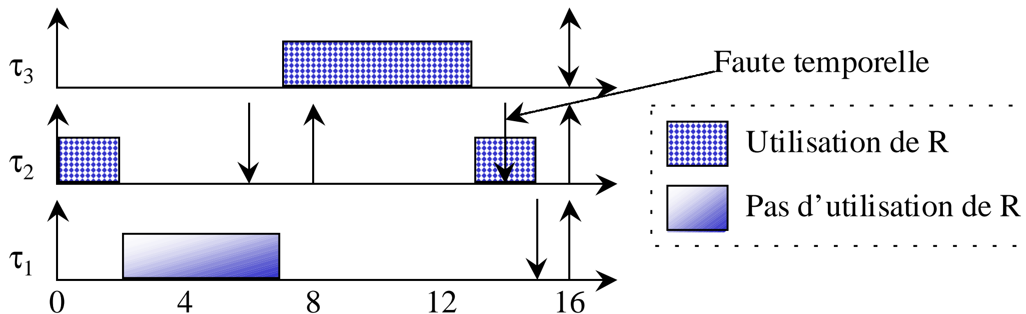


figure I-2-11 : exécution possible du même système

❏

Remarque : aucun algorithme conservatif ne peut ordonnancer ce cas puisqu’il faut absolument pour cela laisser le processeur oisif entre les instants 7 et 8 bien que τ_3 est à ce moment là actif.

L’ordonnançabilité en-ligne d’un système de tâches partageant des ressources critiques ne peut donc plus être décidée directement par simulation puisque la pire durée n’est pas le pire cas.

Il est donc nécessaire d’intégrer dans la durée des tâches le temps de blocage (temps maximal passé à attendre la fin de la section critique d’une tâche de priorité plus faible) qui peut intervenir à l’entrée d’une section critique. La NASA a fait les frais de ce problème lorsque le module Sojourner, faisant partie de la mission *MARS pathfinder*, est arrivé sur la planète Mars [Cot 99]. Ce temps de blocage ne pouvant être borné naturellement, il est nécessaire d’utiliser des protocoles de gestion de ressources. Dans [Mok 83], l’auteur contourne le problème en proposant de choisir les quanta de temps en s’arrangeant pour que les sections critiques soient de durée unitaire, ce qui n’est pas réaliste dans beaucoup de cas. [Kai 82] propose quant à lui le concept de superpriorité, dont la seconde forme est le précurseur du protocole à priorité héritée.

I-2.2.3.a Le protocole à priorité héritée (PPH)

iii) Définition

Le premier protocole proposé [SRL 87][SRL 90] pour palier l'inversion de priorité est le **protocole à priorités héritées** (ou ***PPH***). Son fonctionnement est assez intuitif, puisqu'il affecte à la tâche détenant une ressource bloquante la priorité la plus grande parmi les priorités des tâches qu'elle bloque. A chaque fois qu'elle libère une ressource, on recalcule sa priorité héritée (i.e. la priorité maximale des tâches qu'elle bloque). Elle reprend sa priorité initiale lorsqu'elle ne détient plus de ressource bloquante.

La figure I-2-12 montre comment le problème d'inversion de priorité est résolu par *PPH*. L'algorithme d'ordonnancement choisi pour cet exemple est *DM*.

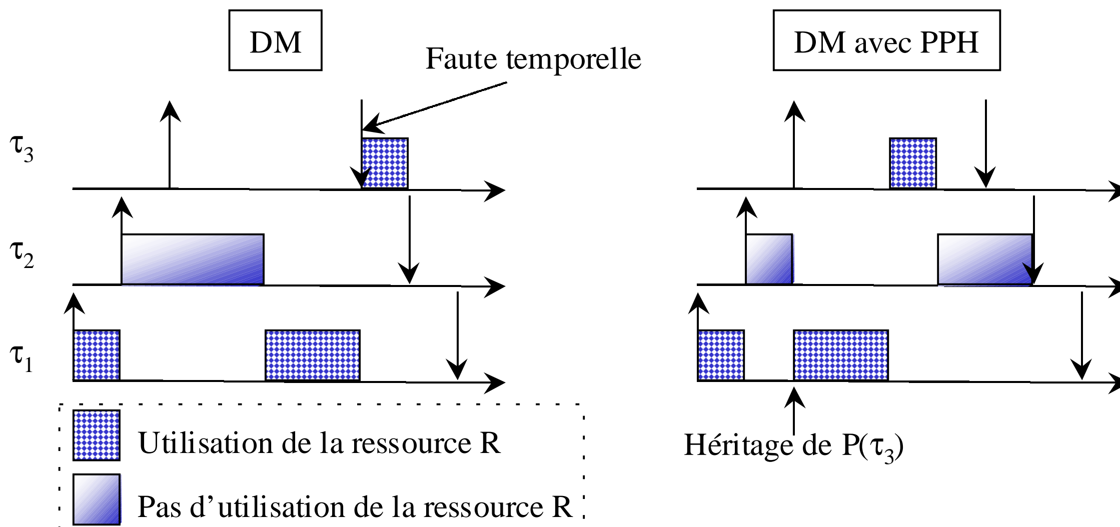


figure I-2-12 : résolution de l'inversion de priorité par le protocole à priorité héritée

Ainsi, une tâche τ_i ne peut rester bloquée par une tâche τ_j de priorité inférieure que si celle-ci est réveillée avant elle et détient une ressource nécessaire à son exécution, ou bien une ressource nécessaire à l'exécution d'une autre tâche bloquée τ_k de priorité supérieure à τ_i. En effet, dans ce dernier cas, τ_j exécute sa section critique avec la priorité de τ_k.

Une tâche τ_i peut donc être bloquée par une tâche τ_j de priorité inférieure si :

– τ_j a commencé sa section critique avant le réveil de τ_i (blocage direct),

ou si

– la ressource utilisée par τ_j lui a fait hériter d'une priorité plus élevée que celle de τ_i : on parle de **blocage transitif**.

Une tâche ne peut être bloquée à l'entrée d'une section critique qu'une fois par ressource existante. Le temps maximal de blocage d'une tâche entrant en section critique est donc donné par la somme des plus longues sections critiques sur chaque ressource à condition que les prises et libérations de ressources ne soient pas entrelacées.

Théorème I-2-27 : [SRL 87][SRL 90] Dans le contexte $|1|r_i,C_i,D_i,P_i,res|$, une tâche τ_i d’un système S ne peut être bloquée que pendant un temps égal à

$$\sum_{R \text{ ressource}} \max_{\tau \in S-\{\tau_i\}} \{\text{durée de l'utilisation de R par } \tau\}$$

iv) Exemple

Soit un système S composé de trois tâches et deux ressources critiques R_1 et R_2, tel que :
S={<0,3,12,12,{<0,3,0><3,0,0>}>,<1,2,8,8,{<2,0,0>,<0,2,0>}>,<2,2,6,6,{<0,2,0>,<0,2,0>}>}

La séquence d’ordonnancement produite par RM avec PPH sur S est donnée sur la figure I-2-13. La tâche τ_3 est bloquée pendant sa première activation par les tâches τ_1 et τ_2 pendant la durée de leurs sections critiques respectives. On peut remarquer que, suivant l’implémentation du PPH, τ_1 pourrait s’exécuter avant τ_2 au moment où τ_3 se réveille puisqu’à cette date, τ_1 et τ_2 ont toutes deux hérité de la priorité de τ_3.

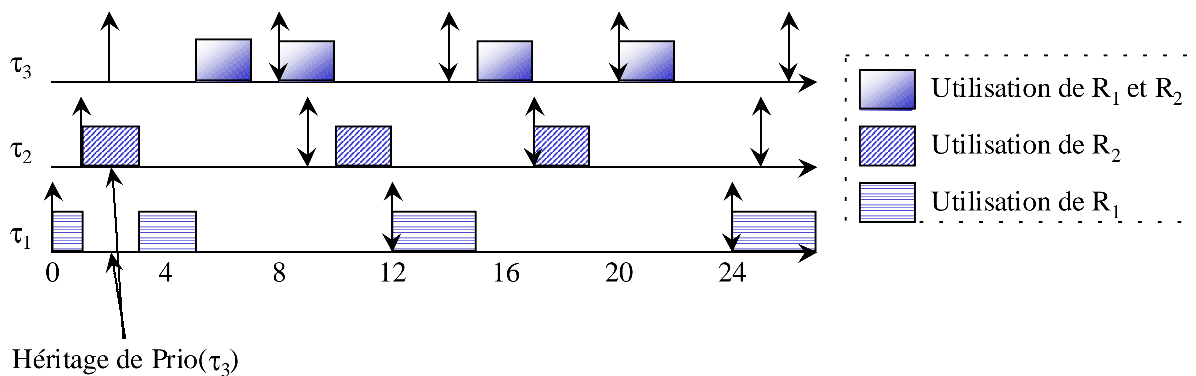


figure I-2-13 : utilisation du protocole à priorités héritées

v) Limitations

Le problème est que l’utilisation de PPH ne réduit pas suffisamment le temps de blocage qui peut être imposé aux tâches. Il faut en effet ajouter une durée supplémentaire aux tâches juste avant leur entrée en section critique qui représente le temps pendant lequel la tâche peut rester bloquée, et ce temps peut être une somme de durées de plusieurs sections critiques, comme c’est le cas sur l’exemple présenté sur la figure I-2-13. En effet, pour chacune des tâches, on doit ajouter la plus longue durée des sections critiques des ressources partagées avec des tâches de priorité inférieure. En plus, il faut prendre en compte les ressources que des tâches de priorité plus faible partagent avec des tâches de priorité plus forte, puisque les tâches pourront dans les sections critiques concernées hériter d’une priorité plus forte. Le calcul du temps de blocage reste donc assez complexe, d’autant plus qu’il doit prendre en compte les inclusions éventuelles de sections critiques. Le calcul des temps de blocage est dans la plupart des cas non triviaux effectué de manière ad hoc. Ce qu’il est important de retenir c’est que la durée de blocage peut être relativement grande (nombre de ressources fois durée maximale des sections critiques des tâches de priorité inférieure) et rendre le système non ordonnançable. De plus, elle ne peut être calculée que si les tâches ont une priorité fixe, donc PPH n’est utilisable qu’avec des algorithmes d’ordonnancement à priorités statiques. Enfin, ce protocole n’empêche pas les interblocages.

Pour améliorer cette lacune du PPH, ses créateurs ont proposé le protocole à priorité plafond.

I-2.2.3.b Le protocole à priorité plafond (PPP)

Le **protocole à priorité plafond**[SRL 90] ou ***PPP*** permet d'éliminer les interblocages, sans passer par des contraintes portant sur l'ordre des demandes de ressources, et d'éliminer les phénomènes où plusieurs sections critiques doivent se terminer avant qu'une tâche de haute priorité ne puisse entrer dans sa section critique, comme sur la figure I-2-13. L'idée principale de ce protocole est de bloquer préventivement des tâches qui peuvent provoquer des inversions de priorités.

i) Définition

A l'origine, *PPP* est destiné par ses auteurs aux algorithmes d'ordonnancement à priorités fixes. Cependant, [CL 90] l'ont étendu à l'algorithme *ED*, le protocole résultant, nommé **protocole à priorité plafond dynamique** ou ***PPPD***, recalcule dynamiquement les priorités plafond du système. Chaque ressource est caractérisée par une priorité plafond, qui est la priorité la plus forte des tâches qui l'utiliseront. Une tâche n'est autorisée à entrer en section critique que si :

1. la ressource demandée est libre, contrainte habituelle, dans le cas contraire, la tâche est bloquée,
2. et sa priorité est strictement supérieure à la plus grande priorité plafond des ressources utilisées par d'autres tâches qu'elle-même. Dans le cas contraire, la tâche est bloquée, on parle dans ce cas de **blocage d'évitement**.

Nous notons système\\$\tau_i$ le système de tâches privé de la tâche τ_i. La priorité plafond de système\\$\tau_i$ représente la plus grande priorité plafond des ressources détenues par des tâches autres que τ_i.

De plus, lorsqu'une tâche τ_i se voit refuser l'accès à une ressource, toutes les tâches détenant une ressource de priorité plafond supérieure ou égale à celle de τ_i héritent de sa priorité, puisque τ_i ne pourra pas entrer en section critique avant qu'elles ne terminent leurs sections critiques respectives.

L'exemple utilisé sur la figure I-2-13 est repris pour illustrer le *PPP* avec *RM* (ou *DM* puisque les tâches sont à échéance sur requête), la séquence d'ordonnancement produite est donnée sur la figure I-2-14.

S={<0,3,12,12,{<0,3,0><3,0,0>}>,<1,2,8,8,{<2,0,0>,<0,2,0>}>,<2,2,6,6,{<0,2,0>,<0,2,0>}>}

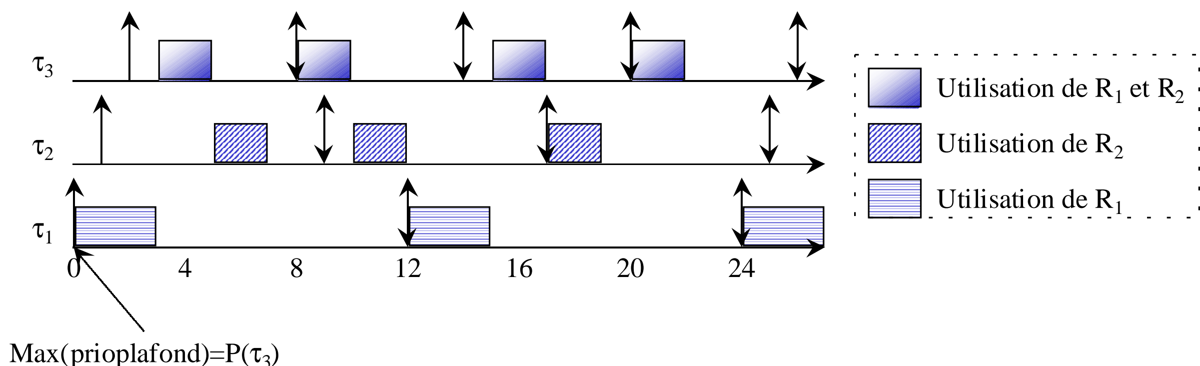


figure I-2-14 : utilisation du protocole à priorité plafond

A l'instant où τ_2 se réveille pour la première fois, la priorité plafond du système\\τ_2 est égale à la priorité de τ_3. τ_2 ne peut donc pas entrer en section critique. Cet exemple montre

que le *PPP* évite à une tâche prioritaire d'attendre la terminaison de plusieurs sections critiques avant de pouvoir entrer dans sa propre section critique.

ii) Propriétés

Le *PPP* garantie l'absence d'interblocage. En effet un interblocage ne peut survenir que si plusieurs tâches accèdent à plusieurs ressources qu'elles détiendront en même temps, mais dans un ordre différent. Pour simplifier, considérons sur la figure I-2-15 deux tâches, τ_1 et τ_2, utilisant deux ressources communes R_1 et R_2. τ_1 prend R_1, mais, avant de demander R_2, elle est préemptée par τ_2 de priorité supérieure qui vient d'être activée. τ_2 prend alors R_2, puis demande R_1. Or R_1 est détenue par τ_1, et τ_1 ne peut pas libérer R_1 avant d'avoir pris R_2, détenue par τ_2. Les deux tâches se bloquent mutuellement : c'est un interblocage.

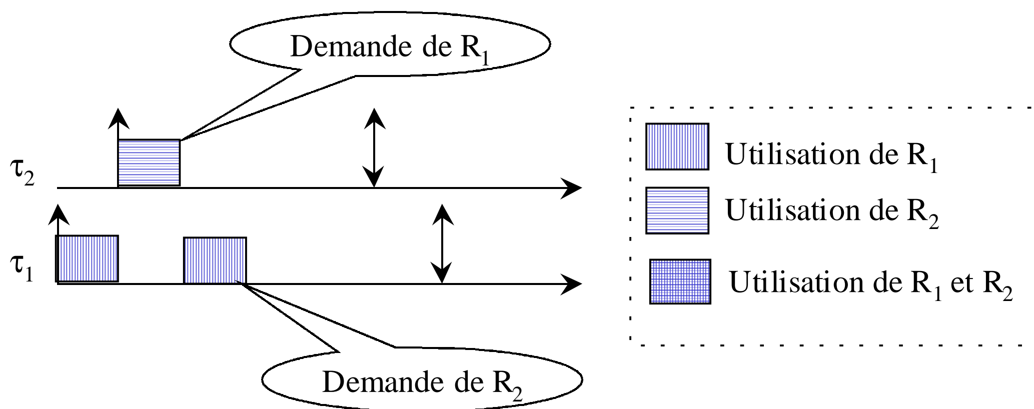


figure I-2-15 : un interblocage

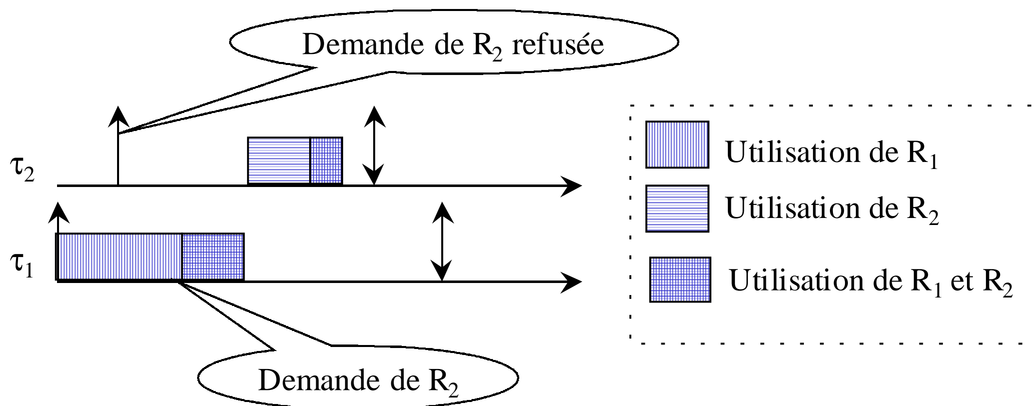


figure I-2-16 : interblocage évité par le PPP

La figure I-2-16 montre qu'avec le *PPP*, l'interblocage est évité. En effet, les deux ressources ont une priorité plafond égale à la priorité de τ_2, qui est plus prioritaire que τ_1. Lorsque τ_2 demande à prendre R_2, elle lui est refusée car la priorité plafond du système$\backslash\tau_2$ est égale à la priorité de τ_2. τ_1 reprend donc la main avec la priorité de τ_2, prend R_2, et termine sa section critique avant que τ_2 ne puisse prendre la main. Une petite particularité rarement détaillée sur le *PPP* est le changement de priorité lors de la libération d'une ressource. Pour simplifier, on dit habituellement qu'une tâche quittant une section critique retrouve la priorité qu'elle avait avant son entrée en section critique, or, on peut imaginer que τ_1 prenne R_1 puis R_2, enfin

libère R_2 puis R_1. Si la tâche τ_2 ne s'était réveillée qu'après que τ_1 ait pris R_2, τ_1 n'aurait hérité de $P(\tau_2)$ qu'après la prise de R_2. Donc à la libération de R_2, τ_1 détient toujours R_1 mais retrouve sa priorité d'origine bien que R_1 soit bloquante pour τ_2. Si une tâche de priorité intermédiaire attend le processeur, il y a un phénomène d'inversion de priorité. La solution consiste donc à associer un niveau de priorité hérité à chacune des ressources détenues par une tâche en plus de sa propre priorité héritée. Lorsqu'elle libère une ressource, sa priorité héritée est recalculée comme étant le maximum des priorités héritées des ressources qu'elle détient.

L'utilisation du *PPP* garantit qu'une tâche τ_i ne peut être bloquée que pendant la durée d'une section critique d'une tâche de priorité inférieure utilisant une ressource de priorité plafond plus grande ou égale à celle de τ_i. En effet, supposons qu'une tâche soit en attente d'une ressource détenue par une tâche de priorité inférieure. La priorité plafond de la ressource concernée est au moins égale à sa priorité, ce qui interdit à toute autre tâche de priorité inférieure de détenir une ressource.

Théorème I-2-28 : [SRL 90] Le **temps de blocage maximal B_i** d'une tâche τ_i est donné par la durée de la plus grande section critique des autres tâches.

Une CS [SRL 90] d'ordonnançabilité d'un système de *n* tâches dans le contexte $|1|r_i,C_i,D_i=P_i,P_i,res|$ (par convention, les tâches sont ordonnées suivant les périodes croissantes) par *RM* avec *PPP* est donnée par :

$$\forall i \in \{1..n\},\ \frac{B_i}{P_i} + \sum_{j=1}^{i} \frac{C_j}{P_j} \leq i.(2^{1/i} - 1)$$

Cette CS est basée sur la CNS de *RM* dans le contexte $|1|r_i=0,C_i,D_i=P_i,P_i|$ (qui est une CS dans le contexte $|1|r_i,C_i,D_i=P_i,P_i|$ (Théorème I-2-7)) : elle considère incrémentalement le système de tâches en commençant par les plus prioritaires. La preuve de ce résultat se fait par récurrence sur le nombre de tâches :

Pas 1 : Si la CS est satisfaite pour un système d'une tâche τ_1 de durée C_1 pouvant être bloquée B_1 unités de temps, alors la CS du contexte $|1|r_i,C_i,D_i=P_i,P_i|$ est satisfaite pour le système de 1 tâche indépendante de durée C_1+B_1 et ce système est ordonnançable.

Hypothèse de récurrence : Supposons que la satisfaction de la CS pour *m* tâches implique que le système de *m* tâches soit ordonnançable.

Pas *m*+1 : Ajoutons une tâche τ_{m+1}, de charge C_{m+1} au système de *m* tâches. Cette tâche peut être retardée par les tâches de priorité inférieure au plus B_{m+1} unités de temps. Le système de *m* tâches vérifié au pas précédent reste ordonnançable car la tâche τ_{m+1} a une priorité inférieure et les retards qu'elle peut leur infliger en utilisant des ressources ont déjà été pris en compte. Si la CS est satisfaite au pas *m*+1, alors la CS du Théorème I-2-7 dans le contexte $|1|r_i,C_i,D_i=P_i,P_i|$ est satisfaite aussi et le système de *m*+1 tâches est ordonnançable.

❑

[CL 90] ont donné une CS d'ordonnançabilité pour *ED* avec *PPPD* dans le contexte $|1|r_i,C_i,D_i=P_i,P_i,res|$ $\sum_{i=1}^{n} \frac{C_i + B_i}{P_i} \leq 1$. Elle reprend la CNS du contexte $|1|r_i,C_i,D_i=P_i,P_i|$ en considérant le cas pathologique où les tâches subissent toutes leur pire durée de blocage.

iii) Limitations

Le *PPP* est donné pour *RM* et *ED* avec une condition suffisante d'ordonnançabilité pour les tâches à échéance sur requête. Si elle n'est pas respectée, ou bien si les tâches ne sont pas à

échéance sur requête, il reste à construire la séquence d'ordonnancement du système en intégrant les durée de blocages maximales à toutes les tâches. Si la séquence construite est valide, alors l'ordonnancement sera valide, puisque les durées au pire prennent en compte les blocages potentiels et la situation de la figure I-2-11 ne peut pas survenir. Il faut souligner que l'intégration des durée de blocages maximales dans les charges des tâches augmente artificiellement la charge des tâches qui peut donc dépasser les *100%* fatidiques.

I-2.2.3.c Le protocole d'allocation de la pile (PAP)

L'adaptation à *ED* du *PPP* étant très coûteuse à cause de la réévaluation des priorités plafond au cours de la vie de l'application, [Bak 91] a proposé un autre protocole particulièrement bien adapté à *ED* et relativement peu coûteux à mettre en œuvre : le **protocole d'allocation de la pile** ou ***PAP***.

i) Définition

Chaque tâche τ_i se voit attribuer un **niveau de préemption** π_i de façon statique. Afin de simplifier les explications suivantes, nous supposons que pour l'algorithme *ED*, les niveaux de préemption des tâches sont assignés suivant les priorités *DM*, c'est à dire de façon inversement proportionnelle au délai critique. Nous insistons sur le fait que cette règle n'est pas obligatoire.

Chaque ressource *R* se voit affecter une valeur plafond C_R, qui est la valeur maximale des niveaux de préemption des tâches actives ayant besoin de plus d'instances de la ressources *R* qu'il n'y en a de disponible. Le plafond d'une ressource est par conséquent dynamique. Le plafond système est alors la valeur maximale des plafonds des ressources en cours d'utilisation.

Une tâche τ_i, qu'elle utilise ou non une ressource, peut préempter une autre tâche τ_j si les conditions suivantes sont vérifiées :

- Son échéance est au moins aussi proche que celle de τ_j (i.e. respect de l'algorithme *ED*),
- $\pi_i > \pi_j$, i.e. le niveau de préemption de τ_i est supérieur à celui de τ_j (ce qui est le cas si les niveaux de préemption sont assignés suivant une priorité *DM*),
- son niveau de préemption est supérieur au plafond système.

Avec ces trois règles relativement simples à mettre en œuvre, une tâche réussissant le test de préemption ne peut pas être bloquée par l'attente d'une ressource : toutes les ressources nécessaires sont disponibles dès le début de son exécution.

ii) Propriétés

Le *PAP* a les mêmes propriétés que le *PPP* : absence d'interblocage, pas d'inversion de priorité, un seul blocage infligé au maximum à une tâche. La priorité est implicitement héritée par une tâche bloquant une tâche de priorité supérieure car la valeur du plafond système prend le niveau de préemption de la tâche bloquée, empêchant toute tâche de niveau de préemption intermédiaire de préempter la tâche bloquante. Ce protocole permet en plus de gérer des ressources avec plusieurs instances.

Le *PAP* tient son nom du fait qu'une tâche préemptée par une autre tâche ne peut pas la préempter à son tour : les tâches préemptées voient leur contexte empilé, et à la fin d'une tâche, c'est soit une tâche juste activée, soit la tâche en sommet de la pile qui est exécutée. En effet, supposons qu'au début de l'application, τ_i soit exécutée. Elle est ensuite préemptée par τ_j, donc $\pi_j > \pi_i$, et le contexte de τ_i est sauvegardé sur une pile. Puis τ_j est préemptée par τ_k (donc $\pi_k > \pi_j$), et voit son contexte empilé au dessus de celui de τ_j. Lorsque τ_k sera terminée, le

sommet de la pile est restauré, en l'occurrence τ_j, et à la fin de cette tâche, c'est encore le sommet de la pile, en l'occurrence le contexte de τ_i, qui sera restauré.

Le fait que les contextes peuvent être sauvegardés et restaurés à partir d'une unique pile facilite l'implémentation du protocole.

[Bak 91] a dégagé une CS d'ordonnançabilité dans le contexte $|1|r_i,C_i,D_i,P_i,res|$ pour *ED* avec *PAP* : $\forall i=1,..,n,\ \frac{B_i}{D_i}+\sum_{j=1}^{i}\frac{C_j}{D_j}\leq 1$.

Ce protocole a été adapté dans [SS 94] afin prendre en compte les contraintes de précédence.

I-2.2.3.d Conclusion sur la gestion de ressources dans les algorithmes en-ligne

Seules des conditions suffisantes d'ordonnançabilité sont disponibles, puisque même la simulation doit prendre en compte les durées de blocage qui peuvent être infligées aux tâches. Il existe des systèmes de tâches, tels celui dont la séquence produite hors-ligne par *RM* sur la figure I-2-10, qui ne sont pas ordonnançables par des algorithmes en-ligne, alors que la séquence produite hors-ligne est valide. C'est un des points qui motive l'utilisation de techniques hors-ligne de génération de séquences valides dans le cas où les tâches partagent des ressources critiques.

De plus, les algorithmes conservatifs n'ont pas toujours un bon comportement face à des systèmes de tâches partageant des ressources.

Théorème I-2-29 : Aucun algorithme conservatif n'est optimal ni dans le contexte $|1|r_i,C_i,D_i,P_i,res|$ ni dans le contexte $|1|r_i,C_i,D_i,P_i,noprmpt|$.

Preuve : la preuve se base sur l'exemple suivant, dont on peut fournir une séquence d'ordonnancement valide non au plus tôt, mais pour lequel aucune séquence au plus tôt n'est valide. Soit $S=\{\tau_1<0,2,4,4\{<0,2,0>\}>,\tau_2<0,1,1,5\{<0,1,0>\}>\}$ un système de tâches partageant une ressource R_1 pendant toute la durée de leur exécution (de façon équivalente, on peut considérer que l'on se trouve dans un contexte non préemptif puisque les tâches ne peuvent pas se préempter l'une l'autre).

Le système *S* est ordonnançable puisque la séquence donnée sur la figure I-2-17 est valide.

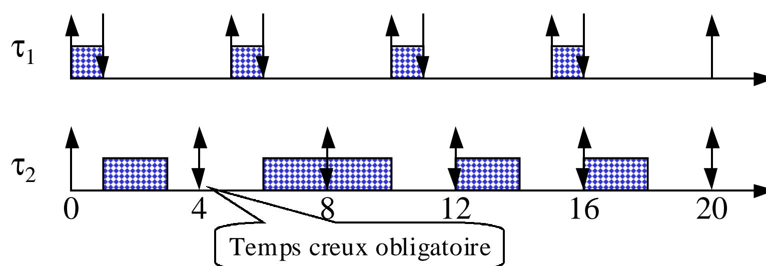


figure I-2-17 : séquence valide pour le système S

Cependant, aucune séquence au plus tôt n'est valide, car comme on le voit sur la figure I-2-18, il est nécessaire que le processeur reste oisif à l'instant 4 alors que τ_2 est active. Dans le cas contraire, la tâche τ_1 ne peut pas respecter son échéance suivante.

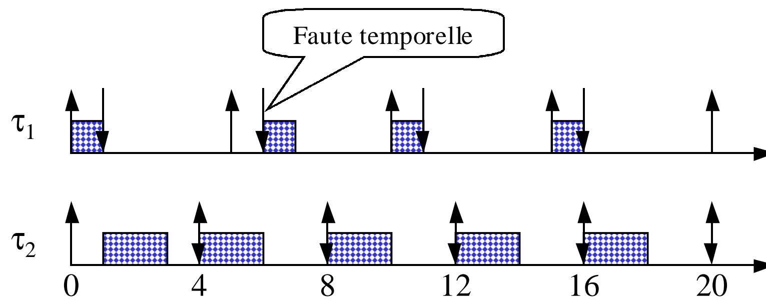


figure I-2-18 : séquence au plus tôt pour le système S

❑

Protocole	Références	Priorités	Ressources multi-instances	Evite les interblocages	Durée des blocages
PPH	[SRL 87] [SRL 90]	Statiques	non	non	Σ des durée des sections critiques
PPP	[SRL 90]	Statiques	non	oui	Plus longue section critique
PPPD	[CL 90]	Variables	non	oui	Plus longue section critique
PAP	[Bak 91]	Variables	oui	oui	Plus longue section critique

Tableau I-2-7 : Récapitulatif sur les protocoles de gestion de ressources

I-2.3 Ordonnancement et complexité en environnement multiprocesseur

Nous avons vu dans les chapitres précédents que des algorithmes polynomiaux optimaux existaient pour certains cas particuliers dans l'hypothèse monoprocesseur sous l'hypothèse préemptive. Les architectures multiprocesseur intéressent de plus en plus la communauté temps réel : si un système de tâches à implémenter est trop coûteux en temps processeur pour que l'on garantisse sa correction temporelle, le concepteur peut se tourner vers une architecture multiprocesseur.

Le modèle multiprocesseur théorique le plus simple à appréhender est le modèle ***PRAM*** [KR 90] pour *Parallel Random-Access Machine* : chaque processeur est identique et a une mémoire privée, mais une mémoire commune est accessible en temps constant par tous les processeurs.

L'avantage de l'architecture *PRAM* par rapport à l'architecture répartie est que le coût de communication entre des tâches placées sur des processeurs différents est négligé, contrairement au cas réparti où cette hypothèse est impossible à émettre [Saa 98][SC 99]. L'hypothèse multiprocesseur néglige donc les aspects optimisation des communications, contrairement au cas réparti, voir [Bea 96][BD 98] pour une étude du placement/ordonnancement dans les systèmes distribués.

Cependant, même si le cas multiprocesseur est plus simple que le cas réparti, la plupart des problèmes d'ordonnancement multiprocesseurs sont NP-complets, et seuls quelques cas parti-

culiers en ordonnancement non périodique ont des solutions optimales pouvant être fournies par des algorithmes polynomiaux.

Dans le cas monoprocesseur, le cas préemptif est plus simple à valider et plus performant que le cas non préemptif. Deux règles font que la différence n'est pas si évidente dans le cas multiprocesseur :

Théorème I-2-30 :[McN 59] Soit un système *S* pris dans le contexte $|m|r_i,C_i|$ dont on veut minimiser la somme pondérée des temps de réponse (i.e. un poids w_i est associé à chaque tâche, et on souhaite minimiser la somme sur *i* de w_i fois temps de réponse de la tâche τ_i). Il existe une solution aussi bonne sous l'hypothèse non préemptive que sous l'hypothèse préemptive.

Ce théorème montre que la différence d'efficacité entre préemptif et non préemptif est moindre que dans le cas monoprocesseur.

Lemme I-2-1 [MD 78][DM 89] : Il ne peut pas exister d'algorithme d'ordonnancement optimal pour $|p\geq 2|r_i,C_i,D_i|$ si les r_i ne sont pas connus a priori

Ce lemme montre qu'aucune approche dynamique ne peut être optimale pour des systèmes de tâches indépendantes à partir du moment où il y a au moins 2 processeurs.

Théorème I-2-31 : Il ne peut pas exister d'algorithme d'ordonnancement dynamique optimal pour $|p\geq 2|r_i,C_i,D_i|$.

Preuve : découle du Lemme I-2-1, car par définition, dans une approche dynamique, les dates de réveil des tâches ne sont pas connues a priori.❑

Nous fournissons deux exemples montrant la non optimalité de *ED* et *LL* qui, rappelons-le, étaient optimaux dans le contexte $|1|r_i,C_i,D_i,P_i|$.

exemple I-2-2 : Soit $S=\{\tau_1\langle 0,3,3,3\rangle,\tau_2\langle 0,1,2,3\rangle,\tau_3\langle 0,1,2,3\rangle\}$ à ordonnancer dans le contexte $|2|r_i=0,C_i,D_i,P_i|$. La figure I-2-19 (a) représente une séquence d'ordonnancement valide pour *S*, et la partie (b) représente la séquence produite par *ED* : celle-ci n'est pas valide.

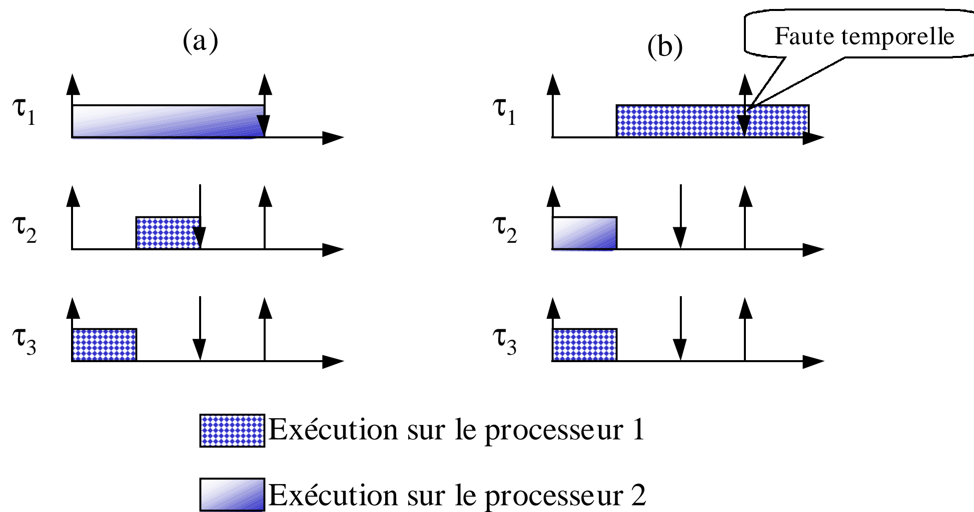


figure I-2-19 : ordonnancement d'un système sur 2 processeurs : (a) est un ordonnancement valide, (b) est produit par ED

Cet exemple montre qu'*ED* n'est pas optimal dès lors qu'il y a plus d'un processeur. Cet exemple, tiré de [DM 89], est ordonnançable par *LL*. *ED* et *LL* n'ont donc plus la même puissance dans un contexte multiprocesseur, cependant, Dertouzos et Mok précisent que *LL* n'est pas optimal non plus dans ce contexte, ce qu'illustre l'exemple suivant.

---

exemple I-2-3 : Soit S={τ_1<0,1,1,4>,τ_2<0,1,2,4>,τ_3<0,2,4,4>,τ_4<2,2,2,4>,τ_5<2,2,2,4>} à ordonnancer dans le contexte $|2|r_i,C_i,D_i,P_i|$. La figure I-2-20 (a) représente une séquence d'ordonnancement valide pour S, et la partie (b) représente la séquence produite par *LL* : celle-ci n'est pas valide.

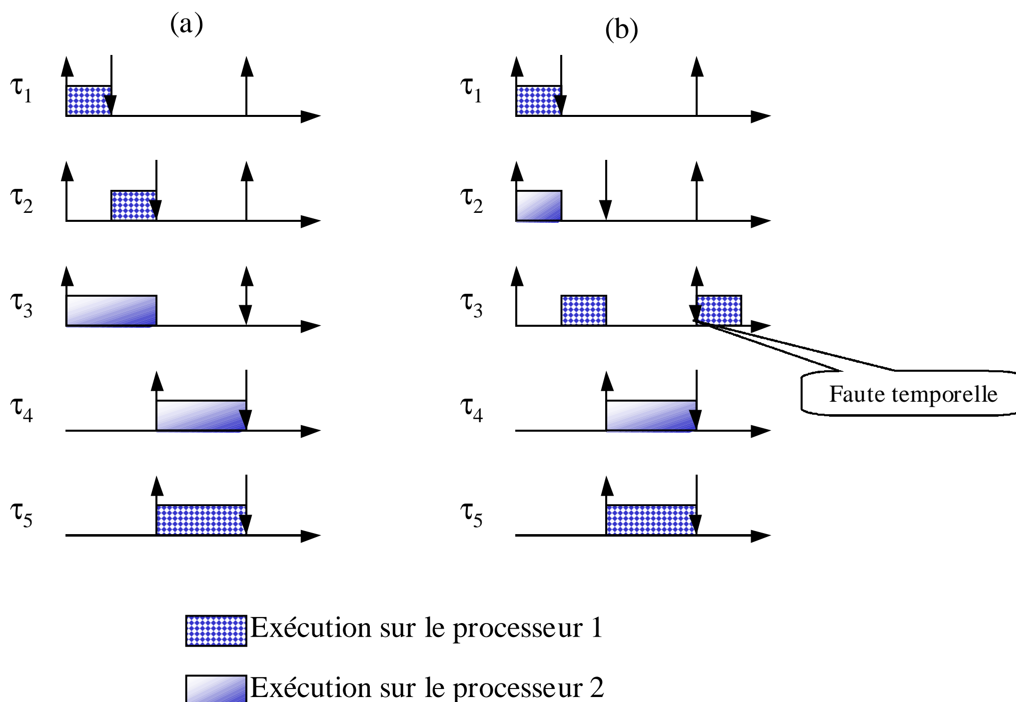


figure I-2-20 : ordonnancement d'un système sur 2 processeurs : (a) est un ordonnancement valide, (b) est produit par LL

Le fait que même les problèmes les plus simples soient impossible à résoudre par des algorithmes dynamiques a conduit les approches dynamiques multiprocesseur à se baser sur les algorithmes en-ligne classiques, en considérant ceux-ci comme des approches *best-effort*, donc non optimales.

Nous renvoyons à [Bla 76] et [SSNB 95] pour une revue détaillée des complexités des problèmes d'ordonnancement multiprocesseur dans les cas particuliers.

I-2.4 Approches hors-ligne

La recherche en techniques d'ordonnancement hors-ligne a donc été motivée par l'absence d'algorithmes polynomiaux optimaux dans le cas général des contextes multiprocesseur même pour les tâches indépendantes.

Dans les contextes monoprocesseur, c'est l'absence d'algorithmes polynomiaux optimaux dès lors que des ressources critiques sont partagées ou bien dès lors que les contraintes de communication ne permettent pas un découpage en forme normale des tâches qui ont motivé les approches hors-ligne.

Les approches hors-ligne, qualifiées aussi d'approches statiques, sont de complexité au moins exponentielle lorsqu'elles sont exactes car il s'agit de produire une séquence d'ordonnancement de longueur exponentielle lorsque les tâches considérées sont périodiques.

On peut classer les approches hors-ligne en deux catégories :

- Les techniques exactes : elles sont basées principalement sur des techniques d'énumération, comme la technique de **séparation et évaluation** (***branch and bound***), ou sur la programmation linéaire en nombres entiers [LM 81][Mar 82], la difficulté, dans ce dernier cas, étant de rendre les contraintes linéaires.
- Les techniques approchées : ce sont des algorithmes sous-optimaux tels que les algorithmes de mise en sac ou de liste, ou bien des algorithmes stochastiques tels que les algorithmes génétiques, ou le recuit simulé. Nous renvoyons le lecteur à [Bea 96] pour un état de l'art détaillé sur ces techniques, qui sont surtout utilisées dans le problème du placement de tâches en environnement réparti.

Ce sont les algorithmes de séparation et évaluation qui sont le plus utilisés en ordonnancement de tâches, puisqu'ils sont simples à mettre en œuvre et parce que l'ajout de contraintes supplémentaires aux tâches (contraintes d'exclusion, de précédence) est beaucoup plus simple à mettre en œuvre que dans le cas de la programmation linéaire en nombres entiers par exemple. Ces approches sont pour la plupart basées sur des systèmes de tâches non périodiques de date de réveil non nulle : ainsi, les auteurs s'intéressant aux systèmes de tâches périodiques, en se basant sur le Théorème I-2-3 (page 31), expliquent qu'il suffit de répéter indéfiniment la séquence produite par l'étude du système entre les instants 0 et *PPCM* des périodes des tâches : chaque tâche périodique τ_j est donc décomposée en $PPCM_{i=1..n}(P_i)/P_j$ occurrences de tâches non périodiques. Il est à noter que cet éclatement des tâches périodiques en tâches non périodiques transforme déjà le problème de façon non polynomiale : tout algorithme hors-ligne travaillant sur un système de tâches non périodiques, lorsqu'il est adapté à un système de tâches périodiques, se voit donc infliger une transformation du système de tâches en entrée en un système de tâches (non périodiques) de taille exponentielle.

De plus, la technique d'éclatement des tâches périodiques en non périodiques n'est valide que si les tâches périodiques considérées sont simultanées, puisque dans le cas contraire (i.e. tâches différées), aucun résultat ne permet de borner le nombre d'instances de chaque tâche à prendre en considération : nous proposerons dans la suite un résultat permettant de prendre en compte les tâches périodiques différées dans toutes les approches hors-ligne prenant en compte les tâches non périodiques.

Les premières approches hors-ligne, relativement similaires, ont été proposées pour le contexte $|1|r_i,C_i,D_i,noprmpt|$ où les auteurs minimisent le retard maximal des tâches, dans [DM 72], [BFR 73] et [BS 74].

Par exemple, dans [BS 74], les auteurs procèdent en construisant un arbre de recherche par énumération des permutations possibles de tâches (ils sont dans un contexte non préemptif) : la racine de l'arbre (niveau 0) est un nœud vide (voir figure I-2-21), possédant n fils, son $k^{ième}$ fils étant étiqueté par la séquence partielle (τ_k). Le fils étiqueté par (τ_k) pourra avoir k–1 fils étiquetés respectivement par $(\tau_k\ \tau_1)$, $(\tau_k\ \tau_2)$,…, $(\tau_k\ \tau_{k-1})$, $(\tau_k\ \tau_{k+1})$,…, $(\tau_k\ \tau_n)$, ces nœuds pourront avoir à leur tour k–2 fils etc. jusqu'à arriver à une profondeur n de l'arbre dans lequel les étiquettes correspondent à une séquence d'ordonnancement complète : en effet, chaque nœud à la profondeur k contient une séquence partielle qui est une permutation (rappelons que l'on est sous l'hypothèse non préemptive) de k tâches. Cette séquence partielle est un préfixe de toute séquence générée à partir de ce nœud.

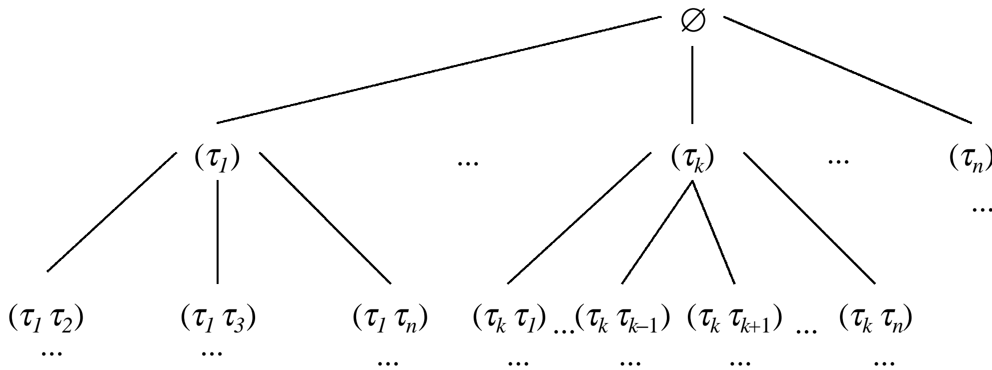


figure I-2-21 : arbre de recherche d'ordonnancement dans le contexte $|1|r_i,C_i,D_i,noprmpt|$

Il y a en tout n! permutations possibles, donc il peut exister en théorie autant de feuilles à l'arbre de recherche, les auteurs ont donc cherché à réduire le temps et l'espace de recherche des solutions via plusieurs heuristiques de recherche.

Avant de construire l'arbre, les auteurs utilisent différentes heuristiques (*ED* non préemptif, *fifo*,…) leur fournissant chacune une séquence d'ordonnancement. La séquence dont le retard maximal est le moins élevé est choisie comme séquence étalon initiale, et la valeur de son retard, notée L_{max}, sera utilisée pour abandonner les nœuds qui ne peuvent mener à une meilleure solution.

Etant donné que *ED* préemptif est optimal dans le contexte $|1|r_i,C_i,D_i|$ pour le critère retard maximal, à chaque nœud construit, on évalue une borne inférieure au retard maximal des séquences d'ordonnancement pouvant être générées à partir de ce nœud en complétant l'ordonnancement déjà construit par les tâches non encore utilisées ordonnancées par *ED* préemptif : cette séquence est forcément au moins aussi bonne que toute séquence non préemptive qui pourra être générée à partir du nœud considéré. Cette borne inférieure, si elle n'est pas meilleure que la borne étalon, indique que le nœud considéré peut être abandonné.

De plus, afin de faire converger plus rapidement l'algorithme vers un optimal, c'est le nœud le plus prometteur (i.e. dont la borne inférieure sur le retard maximal est la moins élevée) qui est développé à chaque pas de l'algorithme, et au fur et à mesure que des séquences complètes sont calculées, la valeur étalon L_{max} est améliorée aussi, permettant d'éliminer davantage de nœuds non prometteurs.

Enfin, lorsque toutes les tâches restant à ordonnancer sont actives, c'est *ED* qui est utilisé puisqu'il est optimal dans le cas $|1|r_i=0,C_i,D_i,noprmpt|$.

Dans [BFR 75], les auteurs se sont intéressés au cas multiprocesseur $|m|r_i,C_i,D_i,noprmpt|$, qui ajoute au problème précédent le problème du placement. Leur méthode de placement est une énumération élégante des placements et ordonnancements des tâches en évitant les symétries triviales comme par exemple « τ_1 sur le processeur 1 et τ_2 sur le processeur 2 » et « τ_1 sur le processeur 2 et τ_2 sur le processeur 1 ». Cette énumération se base sur la construction d'un arbre bi-colore : chaque nœud est soit rond, soit carré, et est étiqueté par un nom de tâche. Si le nœud est rond, alors la tâche en question est placée sur le processeur courant, et si le nœud est carré, alors elle est placée sur un nouveau processeur qui devient le processeur courant.. La racine de l'arbre est vide, et possède n fils étiquetés respectivement par $\tau_1,\tau_2\ldots,\tau_n$, chacun étant un nœud carré signifiant que la tâche considérée est placée sur un nouveau processeur. Ensuite, il existe un chemin d'un nœud N situé à la hauteur h à un nœud N' situé à une hauteur h+1 si :

- τ_i n'est pas déjà dans son préfixe (chemin de la racine à N) car une tâche ne doit être placée qu'une fois,
- τ_i n'est pas dans un carré si il existe dans son préfixe un carré contenant τ_j avec $j>i$, ce qui évite les symétries triviales.
- Il ne peut y avoir plus de m carrés sur un chemin, ce qui assure que le nombre de processeurs nécessaires ne sera pas dépassé.

La figure I-2-22 représente un arbre d'affectation de 3 tâches sur 2 processeurs : le chemin le plus à gauche représente le placement des trois tâches sur un seul processeur, ordonnancées dans l'ordre τ_1 puis τ_2 puis τ_3. Le deuxième chemin en partant de la gauche représente le placement de τ_1 puis τ_2 sur le même processeur, parallèlement au placement de τ_3 sur le second processeur, etc.

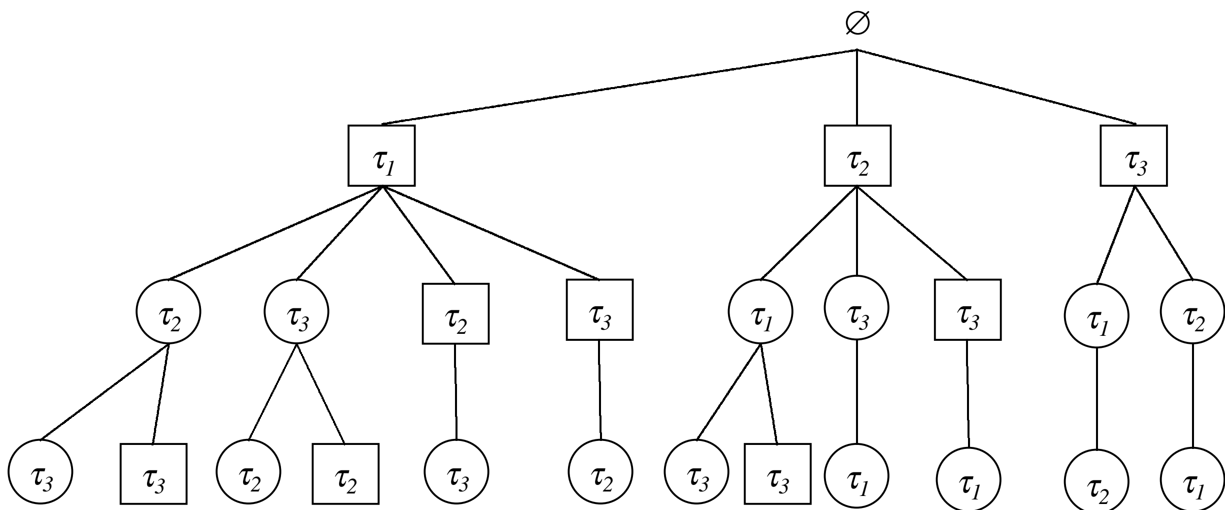


figure I-2-22 : technique d'énumération de placements à l'aide d'un arbre bi-colore

Les méthodes citées résolvent un problème *NP*-difficile par rapport à la taille du système de tâches non périodiques dans les contextes $|1|r_i,C_i,D_i,noprmpt|$ et $|m|r_i,C_i,D_i,noprmpt|$. Rappelons de plus que la transformation d'un système de tâches simultanées périodiques en tâches non périodiques est non polynomiale.

Bien qu'intéressants, ces contextes ne sont que rarement réalistes et les systèmes de tâches temps réel sont souvent composés de tâches communicant et partageant des ressources. Xu et Parnas [XP 90] se sont donc intéressés au contexte non périodique $|1|r_i,C_i,D_i,prec,res|$, qui peut être appliqué, avec un facteur non polynomial, au contexte périodique $|1|r_i=0,C_i,D_i,P_i,prec,res|$.

Leur modèle de tâches est un ensemble de tâches $\tau_1,\ldots,\tau_n$. Une tâche τ_i est composée de C_i segments $\tau_{i,1},\ldots, \tau_{i,Ci}$ qui peuvent être soumis à des contraintes de précédence et des contraintes d'exclusion. On note $\tau_{i,j}\rightarrow\tau_{k,l}$ le fait que le $j^{\text{ème}}$ segment de τ_i précède le $l^{ème}$ segment de τ_k, et $\tau_{i,j}\oplus\tau_{k,l}$ le fait que le $j^{\text{ème}}$ segment de τ_i ne peut être préempté par le $l^{ème}$ segment de τ_k, on nomme cela l'exclusion. Il faut noter que l'opérateur exclusion n'est pas commutatif, et qu'exprimer une exclusion mutuelle entre deux segments $\tau_{i,j}$ et $\tau_{k,l}$ se traduit par deux contraintes d'exclusion : $\tau_{i,j}\oplus\tau_{k,l}$ et $\tau_{k,l}\oplus\tau_{i,j}$.

Leur objectif est de trouver une séquence d'ordonnancement valide minimisant le retard maximal. Pour cela, ils construisent un arbre dont chaque nœud est un ordonnancement pas forcément valide : la racine de l'arbre contient l'ordonnancement donné par *ED*, en tenant compte des contraintes $\oplus$ et $\rightarrow$. A partir de ce nœud, les auteurs dérivent un arbre dont chaque nœud est un ordonnancement meilleur que son nœud père de la façon suivante : la tâche de retard maximal τ_r est repérée dans la séquence, et seul le fait de l'ordonnancer plus tôt pourra améliorer la séquence, ce qui peut être fait si et seulement si :

- on introduit une contrainte de précédence fictive entre le dernier segment de τ_r et un autre segment de tâche d'un ensemble G_1,
- on oblige un segment de tâche d'un ensemble G_2 à être préemptée par τ_r.

Les segments de tâches des ensembles G_1 et G_2 ont les caractéristiques suivantes :

- ils sont exécutés dans l'ordonnancement partiel considéré avant la terminaison de τ_r dans un intervalle où le processeur est continuellement occupé
- leur échéance est plus tardive que celle de τ_r

Donc seul le fait de retarder l'un d'entre eux peut permettre à τ_r d'améliorer sa laxité. Les segments de tâches de l'ensemble G_1 sont ceux qui ne peuvent pas être préemptés par τ_r à cause de contraintes d'exclusion, alors que les segments de G_2 ne sont pas soumis à de telles contraintes.

Un nœud contenant un ordonnancement partiel se voit donc attribuer des nœuds fils de la façon suivante : la tâche tardive τ_r est repérée, les ensembles G_1 et G_2 sont calculés. Pour tout segment s_i de G_1, un fils est créé avec la contrainte supplémentaire dernier segment de $\tau_r\rightarrow s_i$. Pour tout segment s_i de G_2, un fils est créé avec la contrainte supplémentaire τ_r préempte s_i. Un ordonnancement partiel est alors créé pour les nouveaux nœuds par l'utilisation d'un algorithme *ED* prenant en compte les contraintes de précédence, d'exclusion, et de préemption. Un nœud est abandonné lorsque G_1 et G_2 sont vides. Lorsque tout l'arbre est construit, une de ses feuilles contient une séquence d'ordonnancement minimisant le retard maximal du système. On peut aussi arrêter la construction de l'arbre dès lors qu'une séquence avec un retard nul a été trouvée, puisque dans ce cas, la séquence est correcte.

Cette technique permet de prendre en compte des systèmes de tâches relativement contraints, et les auteurs l'ont améliorée afin qu'elle puisse prendre en compte les tâches partageant des ressources entrelacées [XP 92]. Xu a adapté cette méthode dans [Xu 93] au contexte $|m|r_i,C_i,P_i,prec,res|$. Cependant, elles ne permettent pas aisément la prise en compte de contraintes de précédence généralisée (par exemple par boîtes aux lettres), ni le prise en compte de ressources multi-instances ou encore de ressources en lecture-écriture. De plus, rappelons qu'elles ne s'appliquent aux contextes de tâches périodiques que lorsque celles-ci sont simultanées.

I-2.5 Conclusion

Nous avons vu que les approches en-ligne n'étaient optimales que dans certains cas particuliers, mais qu'elles étaient simples à mettre en œuvre. Elles entraînent un surcoût dû à la recherche en-ligne de la tâche prête la plus prioritaire, et au calcul en ligne des priorités pour ce qui est des approches d'attribution variable des priorités. En contrepartie, on les dit plus flexibles car après vérification de conditions suffisantes d'ordonnançabilité, il est certain que l'algorithme d'ordonnancement choisi permettra d'ordonnancer sans faute le système de tâches. De plus, dans le cas où les tâches sont préemptibles et ne sont pas soumises à des contraintes d'exclusion mutuelle, le pire cas pris en compte par l'étude d'ordonnançabilité, notamment en ce qui concerne les durées d'exécution, peut être amélioré en-ligne par la mise en œuvre de l'algorithme. Cependant, dès lors que des parties de tâches ne sont pas préemptibles (en particulier lorsque des ressources critiques sont en jeu), les ordonnancements ne sont pas réguliers, donc on ne peut pas, sans précaution particulière, commencer l'exécution d'une tâche plus tôt que prévu sans risquer de perturber l'ordonnancement. Cela implique que plus il y a de contraintes d'exclusion mutuelle, et moins les conditions suffisantes d'ordonnançabilité par des algorithmes en-ligne sont réalistes.

Les approches hors-ligne paraissent moins flexibles mais permettent de trouver une séquence valide lorsque c'est possible. De plus, leur inflexibilité relative permet l'ordonnancement de systèmes de tâches qui ne pourraient en aucun cas être ordonnancées par des algorithmes en-ligne à cause de la non régularité des ordonnancements lorsque des ressources critiques (ou des parties non préemptibles) sont en jeu.

Jusqu'à maintenant, les techniques hors-ligne ont été utilisées pour trouver des ordonnancements valides qui minimisent le retard maximal. De plus, le contexte le plus général dans lequel elles ont été utilisées est $|m|r_i,C_i,D_i,prec,res|$, ce qui rend les méthodes applicables dans le cadre des tâches périodiques, au contexte $|m|r_i{=}0,C_i,D_i,P_i,prec,res|$.

Dans cette thèse, nous nous intéressons à développer une approche hors-ligne permettant la prise en compte du contexte très général $\boldsymbol{|m|r_i,C_i,D_i,P_i,prmptnoprmpt\ ,prec,resRW|}$, et la recherche de séquences d'ordonnancement ayant d'autres caractéristiques qu'un temps de retard maximal le plus petit possible.

II Extensions des Réseaux de Petri

Dans la suite, une extension de réseau de Petri nous permet de modéliser les systèmes de tâches temps réel et d'extraire des séquences d'ordonnancement valides, il nous faut donc auparavant introduire brièvement les réseaux de Petri et quelques unes de leurs extensions. La gestion du temps étant le critère auquel nous nous intéressons, le modèle choisi doit permettre une prise en compte du temps. Nous présentons donc pour commencer les réseaux de Petri autonomes, puis nous exposons quelques extensions temporelles des réseaux de Petri.

II-1 Réseaux de Petri autonomes

Ce paragraphe ne présente qu'une brève introduction aux réseaux de Petri [Pet 62][HSSM 68] (que nous notons par la suite RdP). Les ouvrages portant sur les RdP sont très nombreux (voir [Pet 80] pour une description des premiers travaux portant sur les RdP) : en ce qui concerne les réseaux de Petri classiques, citons [CV 93] pour une introduction en français, [Pet 81] pour un historique précis des RdP et l'explication des résultats de complexité portant sur les problèmes de base posés sur les RdP, [DE 95] qui expose les propriétés des réseaux de Petri de base lorsque le réseau sous-jacent est à choix libre, et [Mur 89] qui regroupe une bibliographie portant sur une grande partie des travaux traitant des RdP. Ce dernier ouvrage peut être aussi une introduction aux RdP de haut niveau, notamment introduits sous la forme de réseaux colorés par Jensen [Jen 81][Jen 96].

Nous utilisons le terme RdP pour définir ce qui peut être aussi appelé réseau places transitions, notamment dans [BF 86] sur lequel nous nous basons en grande partie pour les notations.

II-1.1 Définitions et notations

Nous donnons ici les définitions et notations usuelles que nous utiliserons par la suite.

Définition II-1-1 : Un **réseau places-transitions** est un quadruplet, $N=\langle Q,T,F,W\rangle$ où :

- $Q = \{p_1,\ldots,p_m\}$ est un ensemble fini de **places**,
- $T = \{t_1,\ldots,t_n\}$ est un ensemble fini de **transitions**,
- $F \subseteq (Q \times T) \cup (T \times Q)$ est le **flot** du réseau,
- $W : F \rightarrow \mathbb{N}^+$ est la fonction de **pondération** du flot ,

Avec $Q \cap T = \emptyset$.

Une place est représentée par un cercle, et une transition par un rectangle. Le flot est représenté par des arcs surmontés de leur pondération. La pondération est omise lorsque le poids d'un arc est unitaire.

Le réseau places-transitions correspondant à la définition $N=\langle\{p_1,p_2\},\{t_1\}, \{\{p_1,t_1\},\{t_1,p_2\}\}, \{\{p_1,t_1\}\rightarrow 1, \{t_1,p_2\}\rightarrow 2\}\rangle$ est donné sur la figure II-1-1.

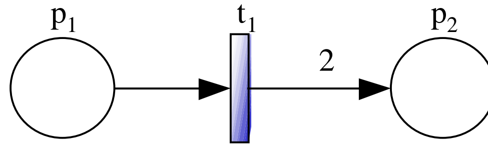


figure II-1-1 : réseau places-transitions

La Définition II-1-1 est une définition statique, ne permettant de définir que le graphe bipartite sous-jacent aux réseaux de Petri. La Définition II-1-3 associe un état initial au réseau, nommé **marquage initial**, et la Définition II-1-4 introduit le comportement du modèle, c'est à dire sa définition dynamique, permettant de passer d'un état à un autre, c'est à dire d'un marquage à un autre.

La Définition II-1-2 introduit le concept d'état d'un réseau de Petri.

Définition II-1-2 : Le marquage du réseau, $M : Q \rightarrow \mathbb{N}$, associe un **marquage courant** au réseau de Petri. On note $M(p)$ le marquage de la place p.

Définition II-1-3 : Un réseau de Petri est un couple $R = \langle N, M_0 \rangle$ où :

- N est un réseau places-transitions, on parle de réseau **sous-jacent**,
- $M_0 : Q \rightarrow \mathbb{N}$ est le **marquage initial**.

Le marquage d'une place est représenté graphiquement par des jetons.

Ainsi, le RdP $R = \langle \{p_1,p_2\},\{t_1\}, \{\{p_1,t_1\},\{t_1,p_2\}\}, \{\{p_1,t_1\}\rightarrow 1, \{t_1,p_2\}\rightarrow 2\} \{3,0\} \rangle$, est représenté graphiquement sur la figure II-1-2.

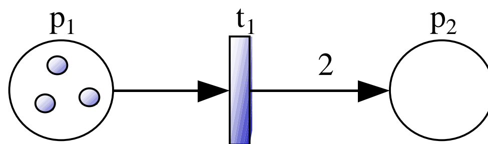


figure II-1-2 :réseau de Petri

La Définition II-1-4 définit la règle de tir (ou franchissement) usuelle des RdP.

Définition II-1-4 : **Franchissement** d'une transition.

- Une transition $t \in T$ est **franchissable** (on dit aussi **sensibilisée**) à partir d'un marquage M si et seulement si $\forall p \in Q$, $M(p) \geq W(p,t)$.
- Le **franchissement** de la transition t **consomme** $W(p,t)$ et produit $W(t,p)$ dans toute place p, conduisant à un marquage M' défini par $\forall p \in Q$, $M'(p) = M(p) - W(p,t) + W(t,p)$.
- On dit alors que le franchissement de t à partir du marquage M mène au marquage M', et on note $M(t\rangle M'$.

Par exemple, la transition t_1 du RdP de la figure II-1-2 est franchissable pour le marquage initial et mène à $M_1 = \{2,2\}$. On peut d'ailleurs construire simplement tout état accessible de ce RdP : $\{3,0\}(t_1\rangle\{2,2\}(t_1\rangle\{1,4\}(t_1\rangle\{0,6\}$, ce que l'on peut noter aussi, si l'on ne s'intéresse pas aux marquages intermédiaires mais seulement à la séquence de transitions franchies, $\{3,0\}(t_1,t_1,t_1\rangle\{0,6\}$.

Un RdP est donc défini de façon statique par sa structure (réseau places transitions) et son état initial (marquage initial). Sa dynamique est donnée par la règle de tir, lui permettant d'évoluer de marquage en marquage.

Définition II-1-5 : L'ensemble de tous les **marquages accessibles** à partir d'un marquage M_0 dans un réseau places-transitions N, noté $Acc(N,M_0)$ est $\{M_i \mid \exists\, t_{i,1},\ldots,t_{i,k},\, M_0(t_{i,1},\ldots,\, t_{i,k}\rangle M_i\}$.

Définition II-1-6 : Une place est **bornée** si pour tout marquage accessible, le nombre de jetons qu'elle contient est borné.

Un réseau de Petri est borné si toutes ses places sont bornées.

Remarque : l'ensemble $Acc(N,M_0)$ peut être représenté sous la forme d'un graphe, non borné si le RdP n'est pas borné, on parle alors de **graphe des marquages** ou **graphe d'accessibilité** : chaque nœud est un marquage du réseau, et il existe un arc étiqueté par une transition t d'un marquage M à un marquage M' si et seulement si $M(t\rangle M'$.

II-1.2 Langage d'un réseau de Petri

Ce qui peut être intéressant, au-delà de l'ensemble des marquages accessibles, c'est l'ensemble des séquences de transitions qui peuvent être franchies, appelé langage du RdP. Si seules certaines transitions sont intéressantes, on peut étiqueter les transitions par des lettres.

Définition II-1-7 : Un **réseau places-transitions étiqueté** est un triplet $<N, \Sigma, \sigma>$ où :

- Σ est un alphabet,
- $\sigma : T \rightarrow \Sigma \cup \{\varepsilon\}$, où ε est le mot vide, est un morphisme d'étiquetage.

Il existe 3 sortes d'étiquetage :

- les **étiquetages libres** (chaque transition est étiquetée distinctement, i.e. la fonction σ est une injection). Généralement, les étiquetages libres considérés n'étiquètent pas les transitions par ε,
- les **étiquetages ε-libres** (i.e. une transition ne peut pas être étiquetée par le mot vide, c'est à dire $\sigma : T \rightarrow \Sigma$),
- les **étiquetages à ε-transitions** permettent à plusieurs transitions d'avoir la même étiquette, y compris ε.

Les mots formés par le parcours du graphe des marquages en remplaçant les transitions par leur étiquette forment le langage du RdP étiqueté.

Définition II-1-8 : Le **langage** d'un réseau de Petri $R=\langle N,M_0\rangle$ étiqueté par $\langle\Sigma,\sigma\rangle$, noté $L(R,\sigma)$, est l'ensemble de tous les mots donnés par les séquences de transitions franchissables de R. Un mot est donné par la concaténation des étiquettes associées aux séquences de transitions pouvant être franchies, i.e.

$$L(R,\sigma)=\{w \mid \exists M\in \mathbb{N}^{|Q|}, t_a,t_b,\ldots\in T, M_0(t_a,t_b,\ldots\rangle M, w=\sigma(t_a,t_b\ldots)\}.$$

Le langage d'un réseau de Petri représente donc l'ensemble des séquences franchissables à partir du marquage initial. Cette définition de langage, contrairement au langage des automates finis, tient compte d'un état initial (marquage initial), mais pas d'un état final. Il peut cependant être intéressant de ne considérer que les séquences de transitions qui mènent à un ensemble de marquages particuliers. En effet, afin de représenter le langage $(ab)^*$, un automate fini est muni d'un état final ; il peut en être de même pour un réseau de Petri, dont on peut vouloir représenter les états terminaux par un ensemble de marquages Ψ.

Définition II-1-9 : Le **langage terminal** d'un réseau de Petri étiqueté est le langage de ce réseau, restreint aux séquences de transitions qui mènent à un marquage de l'ensemble terminal. Il est noté $L(R,\sigma,\Psi)$ où Ψ est un ensemble de marquages terminaux.

$L(R,\sigma, \Psi)=\{w \mid \exists M\in \Psi, t_a,t_b,\ldots\in T, M_0(t_a,t_b,\ldots\rangle M, w=\sigma(t_a,t_b,\ldots)\}$.

$\Psi\subseteq \mathbb{N}^{|Q|}$ est l'ensemble des **marquages terminaux**, généralement donné sous forme de contraintes logiques sur les marquages, on l'appelle donc aussi ensemble des **contraintes terminales**.

Par exemple, l'automate fini donné sur la figure II-1-3 représente le langage $(ab)^*$, le langage (non terminal) du RdP représente le langage $(ab)^*[a]$ (la notation $[a]$ signifie que la lettre a peut être présente ou non) , et son langage terminal est $(ab)^*$.

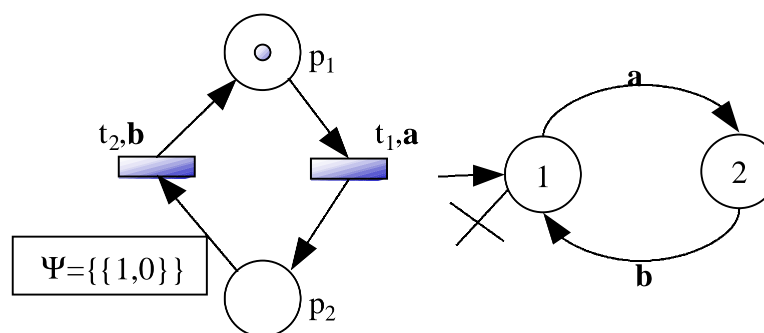


figure II-1-3 : un réseau de Petri et un automate fini dont le langage terminal est $(ab)^*$

La figure II-1-4 montre que le langage terminal permet entre autres de modéliser le langage $a^n b^n$ qui ne peut être représenté par automate fini.

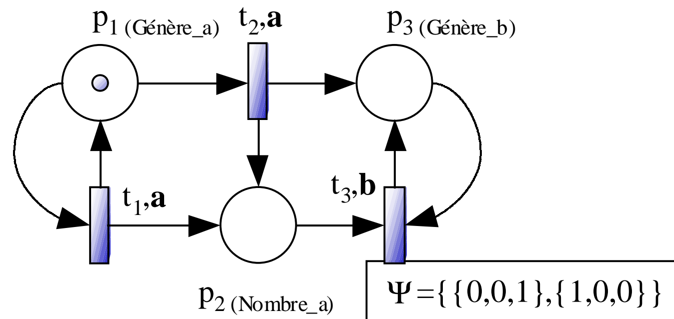


figure II-1-4 : un réseau de Petri dont le langage terminal est $a^n b^n$

Il peut être intéressant de ne s'intéresser qu'à un sous ensemble du langage d'un RdP : ce sous ensemble peut regrouper des mots tels que les états atteints font tous partie d'un ensemble de marquages que l'on juge plus intéressants que d'autres.

Définition II-1-10 : Le **centre $L_C(R,\sigma,\Psi)$ du langage terminal** $L(R,\sigma,\Psi)$ d'un réseau de Petri $R=\langle N,M_0\rangle$ est le sous-ensemble du langage terminal défini par $L_C(R,\sigma,\Psi)=\{w \mid \exists M_1,M_2,\dots M_k \in \Psi,\ t_a,t_b,\dots t_k \in T,\ M_0(t_a\rangle M_1(t_b\rangle\dots(t_k\rangle M_k\ w=\sigma(t_a,t_b\dots,t_k)\}$.

Le centre du langage terminal est donc un sous-ensemble des mots du langage terminal, tels que tous les marquages atteints sont terminaux (ce qui est plus contraignant que dans le cas du langage terminal où seul le marquage final doit être terminal).

II-1.3 Réseaux de Petri colorés

Les réseaux de Petri colorés [Jen 81], inspirés des réseaux prédicat/transition [GL 81][Gen 86], sont une abréviation des réseaux de Petri permettant d'en alléger la représentation graphique. Celle-ci gagne en clarté puisqu'entre autres avantages, la coloration des jetons permet de regrouper plusieurs places en une seule.

Il existe plusieurs définitions des RdP colorés, de plus ou moins haut niveau [Jen 94][Jen 98] suivant le système que l'on souhaite modéliser. La définition que nous utiliserons est la plus simple, car elle ne permet pas aux arcs d'être étiquetés par des variables.

Définition II-1-11 : Un **réseau places-transitions coloré** est un quintuplet, $N=<Q,T,C,F,W>$ où :

- Q, T et F ont la même sémantique que dans les réseaux places-transitions,
- C est un ensemble fini de couleurs,
- $W : F \times C \rightarrow \mathbb{N}$ est la fonction de pondération colorée du flot, on note $W(p_i,t_i,c_i)$ la composante de couleur $c_i \in C$ du poids de l'arc (p_i,t_i).

Le seul changement appliqué au réseau sous-jacent est l'apparition de couleurs sur les arcs, alors que le marquage devient une matrice avec une colonne par couleur :

Définition II-1-12 : Le **marquage coloré** du réseau, est défini par $M : Q \times C \to I\!N$. On note $M(p_i,c_i)$ la composante de couleur $c_i \in C$ du marquage de la place $p_i \in Q$.

La sémantique de la franchissabilité et du tir d'une transition est intuitivement la même que pour les RdP classiques :

Définition II-1-13 : Franchissement d'une transition d'un RdP coloré :

- Une transition $t_i \in T$ est franchissable à partir d'un marquage M si et seulement si $\forall p_j \in Q$, $\forall c_k \in C$, $M(p_j,c_k) \geq W(p_j,t_i,c_k)$.
- Le franchissement de la transition t_i consomme $W(p_j,t_i,c_i)$ et produit $W(t_i,p_j)$ dans toute place p_j, conduisant à un marquage M' défini par $\forall p_j \in Q$, $\forall c_k \in C$, $M'(p_j,c_k)=M(p_j,c_k)-W(p_j,t_i,c_k)+ W(t_i,p_j,c_k)$.

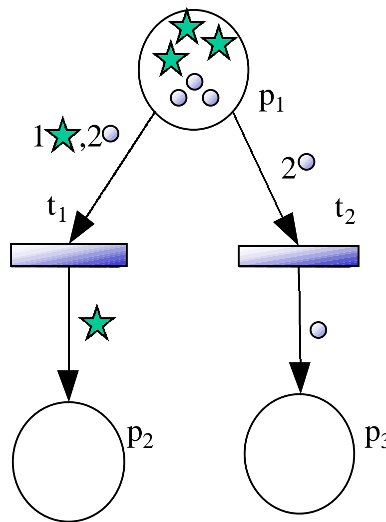


figure II-1-5 : un réseau de Petri coloré

La figure II-1-5 représente un RdP coloré, possédant deux couleurs : les ronds et les étoiles. La transition t_1 nécessite 1 étoile et *2* ronds de p_1 pour produire une étoile dans p_2, alors que t_2 ne nécessite que *2* ronds de p_1 pour produire un rond dans p_3. Remarquons au passage que l'on omet sur la représentation graphique les couleurs qui n'entrent pas en compte dans le tir.

II-2 Réseaux de Petri prenant en compte le temps

Il est important de remarquer que rien n'oblige un RdP autonome à franchir une transition franchissable plutôt qu'une autre, et que ce non déterminisme de franchissement des transitions contribue à la puissance de modélisation des RdP. En effet, la puissance des RdP réside notamment dans le fait que plusieurs actions peuvent être menées en parallèle. Les RdP ont donc naturellement été choisis pour modéliser de nombreux systèmes, et notamment ceux dans lesquels le parallélisme intervient. Cependant, le temps n'est pas pris en compte dans la définition de base des RdP. Cela a amené plusieurs auteurs à considérer des extensions temporelles aux RdP.

II-2.1 Réseaux de Petri temporels

Les **réseaux de Petri temporels** (en anglais time Petri nets) [MF 76] associent un intervalle temporel de sensibilisation aux transitions et une horloge globale au RdP.

II-2.1.1 Définitions et notations

Définition II-2-1 : Un réseau de Petri temporel est un réseau de Petri $R = <N,M_0>$ où le réseau sous-jacent $N = <Q,T,T_{min},T_{max},F,W>$ avec

- $T_{min} : T \rightarrow \mathbb{Q}^*$, $T_{min}(t_i)$ peut être appelée durée minimale de t_i, $\mathbb{Q}^*$ est l'ensemble des rationnels ≥ 0.
- $T_{max} : T \rightarrow \mathbb{Q}^* \cup \{\infty\}$, $T_{max}(t_i)$ peut être appelée durée maximale de t_i.
- $\forall t_i \in T$, $T_{max}(t_i) \geq T_{min}(t_i) \geq 0$.

L'intervalle $[T_{min}(t_i), T_{max}(t_i)]$ est appelé **intervalle de sensibilisation** de t_i.

Graphiquement, l'intervalle de sensibilisation d'une transition se représente par deux nombres entre crochets. La règle de franchissement des transitions impose que le tir d'une transition t_i ait lieu entre $T_{min}(t_i)$ et $T_{max}(t_i)$, c'est à dire que :

- t_i ne peut pas être tirée avant que $T_{min}(t_i)$ unités de temps ne se soient écoulées depuis l'instant où elle est devenue franchissable au sens usuel,
- le réseau ne peut pas manquer exprès l'intervalle de sensibilisation de t_i, et doit tirer celle-ci avant que $T_{max}(t_i)$ unités de temps ne se soient écoulées, sauf bien sûr si une autre transition en conflit avec t_i a consommé les jetons qui étaient nécessaires au franchissement de cette dernière.

En résumé, une transition t_i sensibilisée au sens usuel à l'instant t doit être franchie à une date notée $\theta(t_i)$ avec $t+T_{min}(t_i) \leq \theta(t_i) \leq t+T_{max}(t_i)$ si les jetons nécessaires ne sont pas consommés avant par une autre transition. Le tir d'une transition est instantané.

Les réseaux de Petri temporels ont été initialement proposés dans le but de modéliser des systèmes avec récupération d'erreurs. Leur analyse était rendue délicate par le fait qu'il existe un nombre de comportements infinis étant donné que le nombre de couples possibles <Marquage du réseau, Date de tir d'une transition> est infini pour les dates de tir dans $\mathbb{Q}^*$. Dans [BM 82], une méthode d'étude exhaustive des comportements possibles des RdP temporels

basée sur les classes de marquages est proposée, ce qui a conduit à l'extension de l'utilisation de ce modèle aux systèmes asynchrones dans lesquels le temps a de l'importance, notamment dans le domaine de la qualité de service et l'évaluation de performance des protocoles de communication avec récupération d'erreurs, comme le protocole du bit alterné étudié dans [MB 83][BD 91].

Le **graphe de classe** est composé de couples <Marquage du réseau, Contraintes d'intervalles sur la sensibilisation des transitions franchissables>.

La figure II-2-1 représente la modélisation par RdP temporel d'un système composé d'un livreur de lait, modélisé par la place *Livreur*, passant entre 4 et 6 heure du matin pour déposer du lait sur le paillasson d'une maison. La maison est occupée par un dormeur qui se réveille entre 5 et 10 heure du matin, puis qui peut, sans contrainte de temps, prendre le lait sur le paillasson afin de le boire. Cependant, le lait ne doit pas rester plus de 2 heures au soleil, sinon il caille et n'est plus buvable.

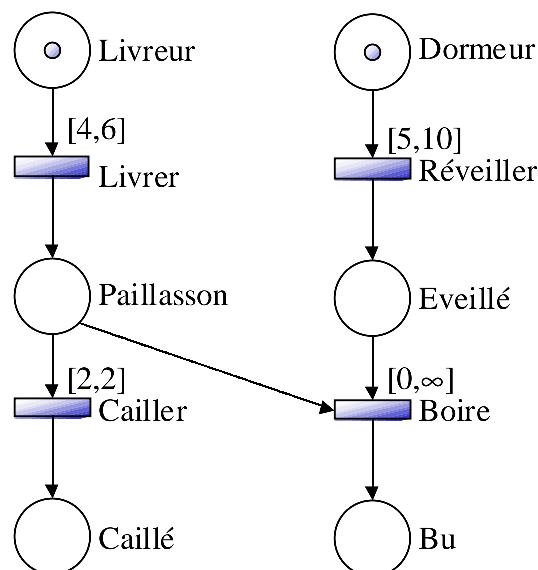


figure II-2-1 : un réseau de Petri temporel

Le graphe des classes permet d'avoir un aperçu de l'ensemble des états possibles du système.

Sur la figure II-2-2, seuls les places contenant un marquage non nul sont représentées, et dans une classe donnée, l'intervalle dans lequel $\theta(t_i)$ est contraint représente les dates de tir possibles de t_i. La classe C_0 exprime le fait que la transition *Livrer* peut être tirée au plus tôt dans 4 heures, et au plus tard dans 6. Parallèlement, *Réveiller* peut tirer au plus tôt dans 5 heures, et au plus tard dans 10. Le tir à partir de C_0 de *Livrer* mène à la classe C_1, dans laquelle *Cailler* tirera dans exactement 2 heures, alors que *Réveiller* tirera au plus tôt dans 1 heure (cas où *Livrer* a tiré au plus tôt), et au plus tard dans 6 heures (cas où *Livrer* a tiré au plus tard). Le tir à partir de C_0 de *Réveiller* mène à la classe C_2, dans laquelle *Livrer* peut tirer au plus tôt tout de suite (en effet, au moins 5 heures se sont écoulées depuis C_0 puisque *Réveiller* a tiré) et au plus tard dans 1 heure (correspond au cas où *Réveiller* a tiré au plus tôt). Ainsi de suite pour les classes, qui sont composées de marquages, et de contraintes sur les dates de tir des transitions franchissables t_i de la forme $\alpha_i \leq \theta(t_i) \leq \beta_i$ et $\theta(t_i)-\theta(t_j) \leq \gamma_{ij}$, avec α_i, β_i, γ_{ij} constantes rationnelles, ce qui rend le système d'inéquations solvable en un temps polynomial.

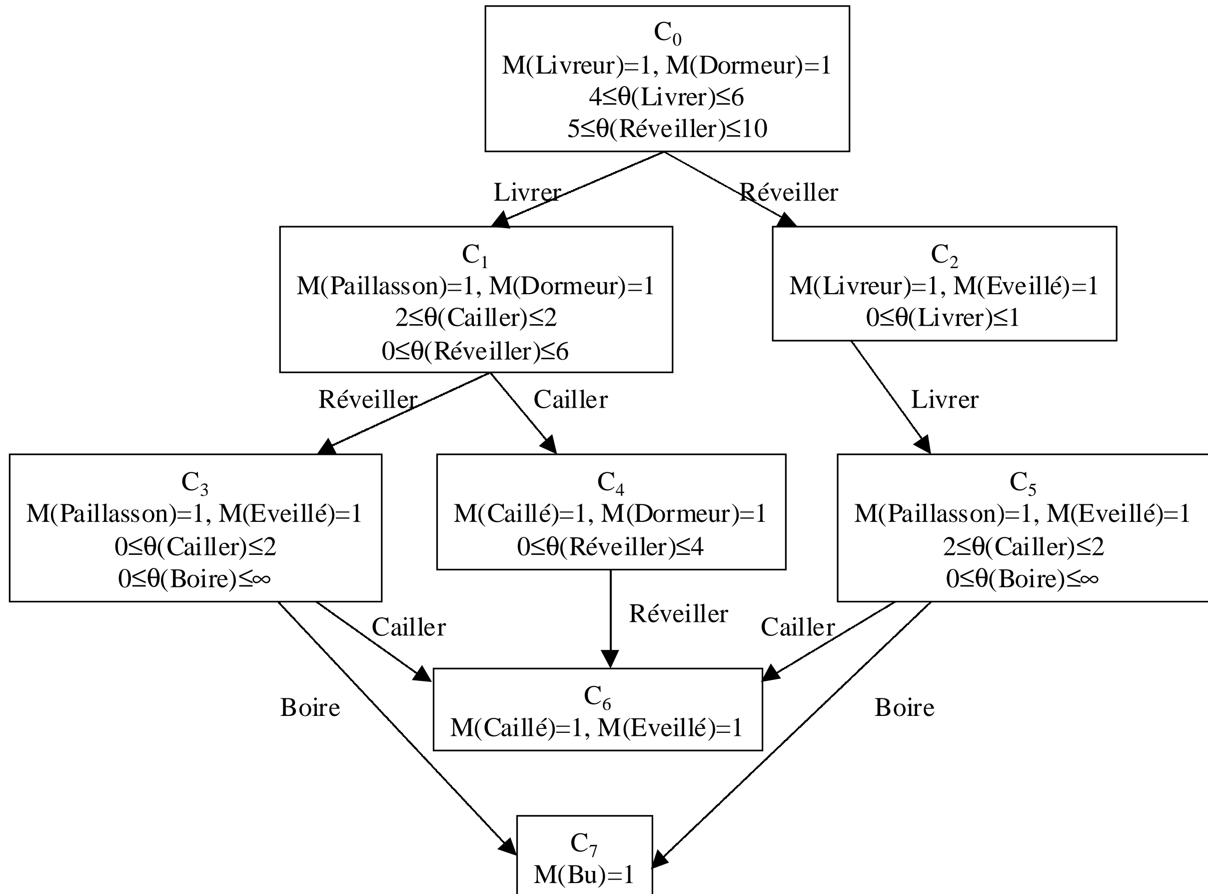


figure II-2-2 : graphe de classe d'un réseau de Petri temporel

Un mot du langage d'un RdP temporel est donné par une séquence de transitions estampillées par leur date de tir. Le langage d'un RdP temporel peut donc être donné par les chemins du graphe des classes où l'estampillage d'une transition allant d'une classe à une autre respecte toutes les inéquations de la classe de départ.

Ce modèle s'avère bien adapté à l'étude de systèmes indéterministes temporisés, lorsque la connaissance de la durée des événements n'est que partielle. Le problème est que cet indéterminisme subsiste dans l'étude des RdP temporels : les bornes temporelles autour du tir d'une transition franchissable t_i ont tendance à s'élargir au fur et à mesure que d'autres transitions sont tirées pour mener vers d'autres classes.

Cependant, lorsque les durées d'exécutions sont effectivement non déterministes, ce modèle est bien adapté à la modélisation.

II-2.1.2 Puissance d'expression

Le principal avantage de ce modèle est sa puissance de représentation. En effet, les RdP temporels ont la puissance de modélisation d'une **machine de Turing** (voir annexe I-2). En effet, ils permettent d'effectuer des tests à zéro et de manière équivalente de modéliser des priorités entre les transitions. Une machine de Turing est un modèle bien connu d'abstraction de la machine de Van Neumann, permettant de modéliser tout programme informatique. Elle est composée d'un ruban infini dans les deux sens, décomposé en cases pouvant être blanches ou bien contenir un symbole, d'une tête de lecture/écriture, d'un état courant, et d'un jeu d'instructions donné sous la forme d'un automate fini dont les états sont l'ensemble des états possibles. L'automate passe d'un état à un autre en fonction du symbole lu par la tête, et ce faisant effectue une action associée à la transition. Une action se compose d'une écriture de la

tête et d'un mouvement de celle-ci. Tout modèle aussi puissant qu'une machine de Turing est capable de modéliser n'importe quel programme informatique. Afin de montrer qu'un modèle de RdP peut modéliser tout programme informatique, il suffit donc de modéliser avec celui-ci une machine de Turing. Le problème est que le modèle de Turing est assez éloigné des RdP, et il est fastidieux de modéliser une machine de Turing. Comme dans [VV-N 81], nous utilisons donc plutôt un autre modèle théorique équivalent aux machines de Turing : les machines à registres [Min 67].

Définition II-2-2 : Une **machine à registres** est un modèle basé sur le modèle informatique composé de :

- r registres $R=\{r_1,\dots,r_r\}$, contenant chacun un nombre entier positif,
- un ensemble d'états $Q=\{q_0,\dots,q_f\}$,
- un programme est un automate fini dont les états sont les éléments de Q. Il possède un état initial unique q_0 et un état final unique q_f. Les liens partant d'un état $q_i \neq q_f$ sont soit au nombre de 1, dans ce cas le lien est étiqueté par $r_j=r_j+1$ pour $r_j \in R$, soit au nombre de *2*, dans ce cas, l'un est étiqueté par *si* $r_j=0$, l'autre est étiqueté par *sinon* $r_j=r_j-1$.

Une machine à registres a une puissance d'expression équivalente à celle d'une machine de Turing.

Puisqu'un état a au plus deux arcs sortant, l'automate peut être aisément décrit de façon textuelle sous la forme

q_i : $r_j := r_j+1$: goto q_k,

q_i : si $r_j=0$ alors:goto q_k sinon $r_j := r_j-1$:goto $q_{k'}$

Théorème II-2-1 : Les réseaux de Petri temporels ont l'expressivité d'une machine de Turing.

Preuve : La preuve de ce théorème est donnée par construction d'une machine à *n* registres à l'aide d'un RdP temporel. La figure II-2-3 représente une machine à *2* registres modélisée par un réseau de Petri. Le programme exécuté est :

```
q0 : si r2=0 alors:goto qf sinon:r2:=r2-1; goto q1
q1 : r1:=r1+1:goto q0
qf :
```

Ce qui pourrait s'exprimer dans un langage de haut niveau par $r_1:=r_1+r_2$ puis $r_2:=0$.

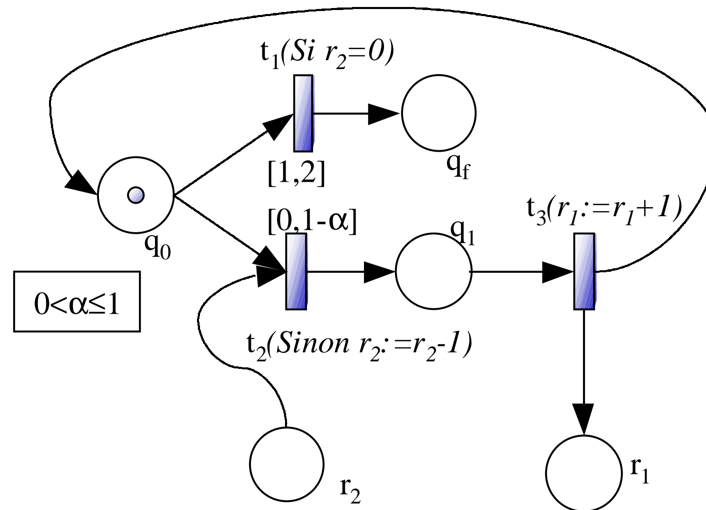


figure II-2-3 : une machine à 2 registres modélisée par RdP temporel

La puissance de modélisation des RdP temporels vient du fait que l'on peut forcer un test à 0 de la façon décrite sur la figure II-2-3 pour les transitions en conflit structurel t_1 et t_2. En effet, de par les intervalles de sensibilisation temporelle imposés à t_1 et t_2, ces deux transitions ne sont jamais en conflit. Si la place r_2 contient un jeton, c'est la transition t_2 qui tire car son intervalle temporel l'oblige à le faire dans la première unité de temps. C'est donc seulement si t_2 n'a pas tiré (c'est à dire si r_2 ne contenait pas de jeton) que t_1 tire. Les temporisations permettent donc d'instaurer une priorité entre des transitions en conflit structurel.

Lorsque des informations sur la répartition probabiliste des durées de tir d'une transition sont disponibles, il peut être intéressant de les intégrer au modèle. Par exemple, sur la figure II-2-1, il est probable que le dormeur se réveille aux alentours de 7h30. Cela peut être pris en compte par la représentation d'une distribution suivant une gaussienne entre 5 et 10 heures avec la probabilité maximale à 7h30. L'extension des réseaux de Petri temporels par des lois probabilistes donne les réseaux de Petri temporels stochastiques [Jua 99][Gal 97].

II-2.2 Réseaux de Petri temporisés

Les réseaux de Petri temporisés (en anglais *timed Petri nets*), introduits dans [Ram 74] et étudiés intensivement dans [Chr 83], associent une durée fixe aux transitions : on parle alors de **réseaux T-temporisés**. Des variantes, de puissance d'expression équivalente, utilisent des temporisations sur les places (on obtient alors des **réseaux P-temporisés**), ou des temporisations mixtes. Nous nous intéresserons dans cette partie au réseaux T-temporisés qui ont été particulièrement utilisés en ordonnancement cyclique [CCG 84][CC 88]. Le lecteur intéressé par d'autres formes de RdP temporisés peut se référer à [BR 90][Zub 91].

La durée d'une transition se réfère à la durée nécessaire à son franchissement. Le tir d'une transition t_i de durée d_i a lieu en deux phases : les jetons nécessaires au franchissement de t_i sont retirés, puis, au bout de d_i unités de temps, les jetons produits par le tir de t_i sont déposés.

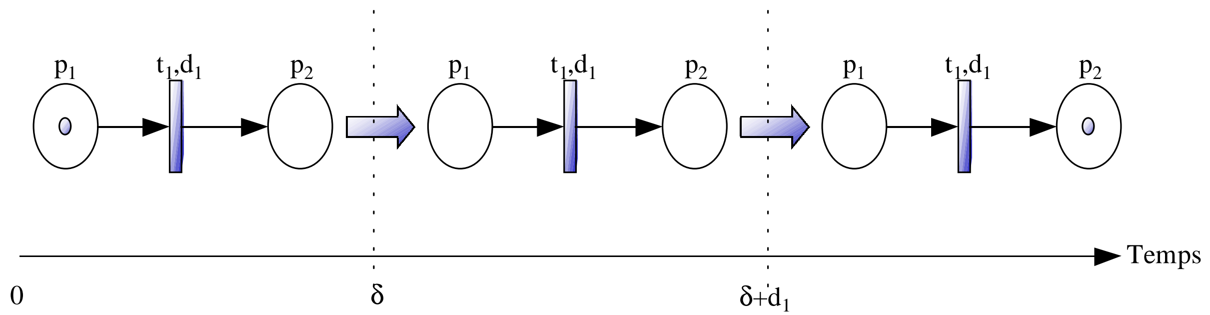


figure II-2-4 : états successifs d'un RdP T-temporisés

La figure II-2-4 représente un RdP T-temporisé comportant une transition t_1, de durée d_1. A l'instant δ, date à laquelle on choisit arbitrairement de tirer t_1, le jeton de p_1 est consommé, et est produit dans p_2 à l'instant $\delta+d_1$. Remarquons que le marquage du RdP n'est pas suffisant pour connaître son état entre les instants δ et $\delta+d_1$: en effet, dans cet intervalle, il nous faut savoir quelles transitions sont en cours, et pour combien de temps encore. Il faut donc coupler au marquage un ensemble de couples {transition, temps restant avant la fin}. De plus, si l'on considère qu'une transition peut être réentrante, c'est à dire qu'elle peut être simultanément en cours de tir plusieurs fois, alors l'ensemble de couples est un multi-ensemble de couples.

Il est possible de considérer deux modes de fonctionnement de RdP T-temporisés : classique, ou bien à **vitesse maximale**. Dans ce dernier cas, une transition doit être tirée dès lors qu'elle est sensibilisée, avec respect de la règle des conflits. Ainsi, avec cette règle de tir, sur la figure II-2-4, δ vaut forcément 0.

Théorème II-2-2 : Un RdP T-temporisé fonctionnant avec la règle de tir à vitesse maximale a l'expressivité d'une machine de Turing.

Preuve : Comme pour les RdP temporels, la preuve est donnée par construction d'une machine à registres sur la figure II-2-5. Celle-ci modélise le programme :

```
q0 : si r2=0 alors:goto qf sinon:r2:=r2-1;goto q1
q1 : r1:=r1+1:goto q0
qf :
```

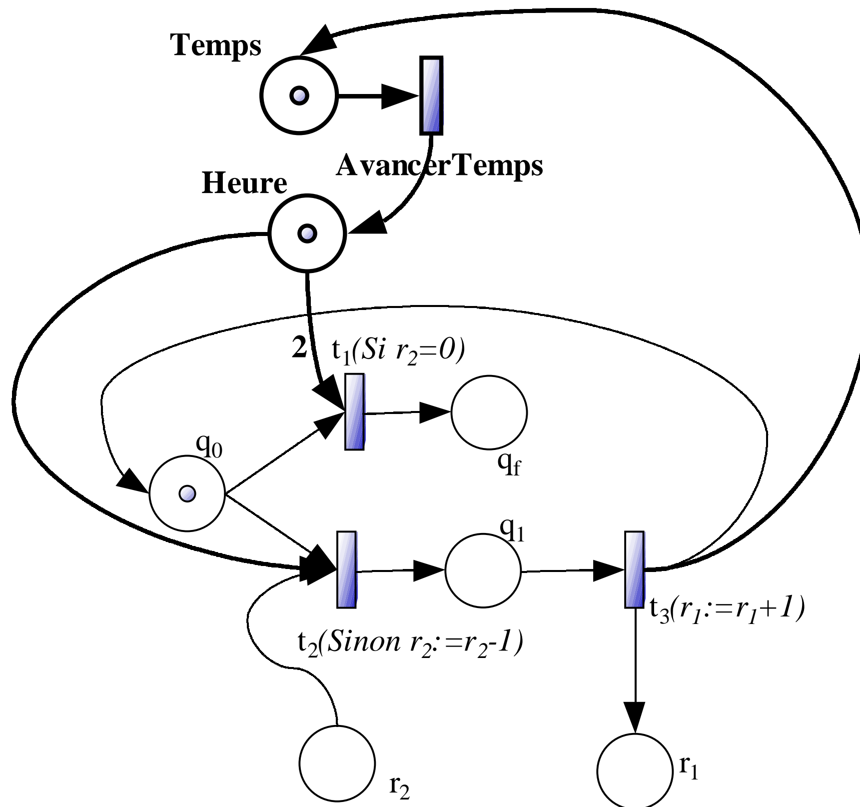


figure II-2-5 : modélisation d'une machine à deux registres à l'aide d'un RdP T-temporisé

Elle représente en traits fins la même structure de machine à registre que le réseau utilisé pour les RdP temporels sur la figure II-2-3. Cependant, chacune des transitions a une durée unitaire (non représentée graphiquement) : il faut donc utiliser un stratagème afin de favoriser la transition t_2 par rapport à la transition t_1. Cela est fait par l'adjonction en gras d'un système d'horlogerie :

- Si r_2 contient au moins un jeton, à l'instant 0, seules les transitions t_2 et *AvancerTemps* sont franchissables. Etant donné que le fonctionnement est à vitesse maximale, elles sont toutes les deux tirées. A la date 1, il y a donc 1 jeton dans la place heure, et un jeton dans q_1. Le registre r_2, quant à lui, a été décrémenté. Seule t_3 est franchissable, ce qui mène à l'instant *2* au marquage initial pour les places autres que les registres, puisque r_2 a été décrémenté, et r_1 a été incrémenté. Le processus peut alors recommencer tant que r_2 n'est pas vide.
- Si r_2 est vide, seule la transition *AvancerTemps* peut être franchie, et doit l'être (étant donné le fonctionnement à vitesse maximale du RdP) : à la date 1, la place *Heure* contient donc *2* jetons, et seule la transition t_1 est franchissable, ce qui est fait. A la date *2*, le RdP est bloqué avec un jeton dans l'état q_f.

❑

La règle de tir à vitesse maximale fournit donc une puissance d'expression supplémentaire aux RdP temporisés. De plus, le fait de se contenter de transitions de durée unitaire dans la modélisation d'une machine à 2 registres montre que la puissance d'un RdP temporisé dont toutes les transitions sont unitaires est la même que celle d'un RdP dont les transitions peuvent être de durée arbitraire. Cependant, le fait de limiter le modèle à des transitions unitaires simplifie la représentation : en effet, il est inutile de noter les temporisations sur les transitions. De plus, cette limitation simplifie grandement l'implémentation car dans ce cas il n'est pas nécessaire de stocker le temps restant avant la fin du tir d'une transition.

Cette limitation équivaut aux RdP avec la règle de tir maximal.

II-2.3 Réseaux de Petri autonomes avec la règle de tir maximal

Les RdP avec la **règle de tir maximal** ont la même structure que les RdP autonomes. Cependant, la règle de franchissement des transitions est différente.

> Définition II-2-3 : La **règle de tir maximal** est définie de la façon suivante : soit $F=\{t_{i_1},t_{i_2},\ldots,t_{i_k}\}$ l'ensemble des transitions franchissables (dont certaines peuvent être en conflit), alors les ensembles de transitions à franchir simultanément $f \subseteq F$ sont tels que pour toute transition $t_i \in F \setminus f$, t_i est en conflit avec au moins une transition de f.

La règle de tir maximal impose donc au RdP de fonctionner comme à vitesse maximale pour les RdP temporisés. Tout se passe comme si le RdP était un RdP temporisé à vitesse maximale dont toutes les transitions sont de durée unitaire. On peut aussi voir un RdP autonome avec la règle de tir maximal comme un RdP temporel dont toutes les transitions sont étiquetées par [1,1] et que le fonctionnement est synchrone.

Par exemple, sur la figure II-2-6, l'ensemble F des transitions franchissables est $F=\{t_1,t_2,t_3\}$. Les ensembles de transitions qui peuvent être franchis sont $f_1=\{t_1,t_3\}$ ou bien $f_2=\{t_2\}$.

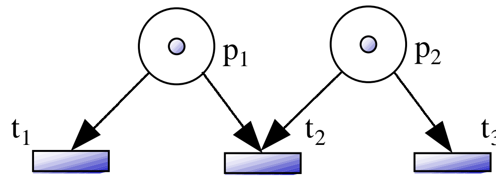


figure II-2-6 : un réseau de Petri avec conflits

L'avantage de l'utilisation de ce modèle par rapport au modèle temporisé est sa simplicité de représentation et surtout d'implémentation. De plus, il en a la même expressivité (i.e. tout RdP T-temporisé à vitesse maximale peut être représenté par un RdP avec la règle de tir maximal). [Sta 90] montre comment passer de façon équivalente d'un RdP T-temporisé à un RdP avec la règle de tir maximal (voir figure II-2-7). Le lecteur peut se référer à [JLKD 86] pour une étude des propriétés de ce modèle.

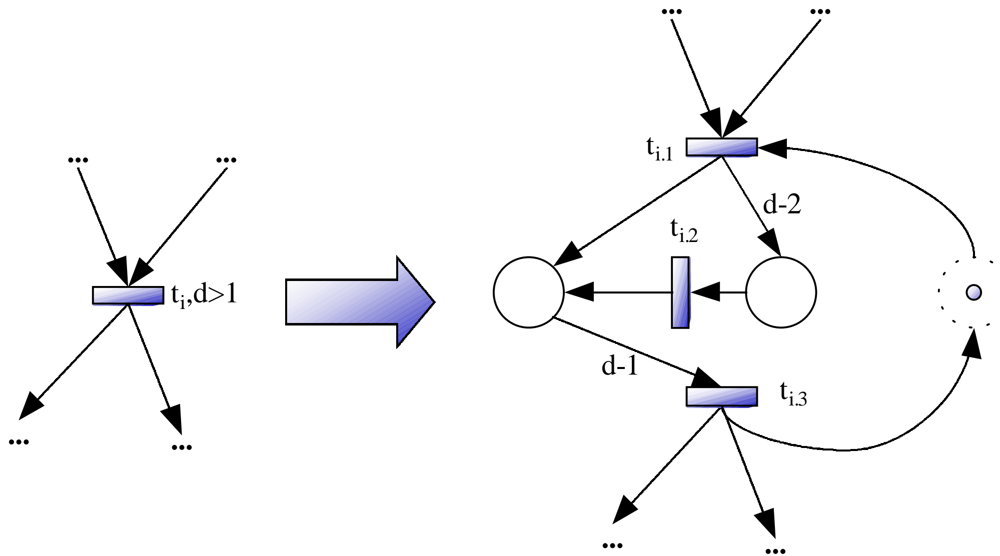


figure II-2-7 : transformation d'un RdP T-temporisé à vitesse maximale en RdP autonome avec la règle de tir maximal, la place en pointillés est utilisée si l'on souhaite éviter la réentrance

Par hypothèse, les transitions sont non réentrantes, ce qui signifie que sur la figure II-2-8, la transition t_1, bien qu'elle puisse être franchie simultanément *2* fois, n'est franchie qu'une seule fois. Les états accessibles de ce RdP sont $\{2\}(\{t_1\})\{1\}(\{t_1\})\{0\}$, et non pas $\{2\}(\{t_1,t_1\})\{0\}$: il faut deux tirs successifs pour vider la place p_1.

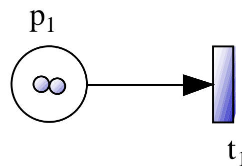


figure II-2-8 : un réseau de Petri simple avec la règle de tir maximal

II-2.4 Conclusion

Lorsque le système modélisé par réseau de Petri nécessite la prise en compte du temps différentes alternatives se présentent. Si les durées à modéliser sont déterministes et fixées, les RdP temporels ne paraissent pas être la meilleure solution, en effet, ils sont surtout adaptés aux systèmes non déterministes. Les réseaux de Petri temporisés semblent bien adaptés à la modélisation de systèmes composés d'actions de durées différentes, mais si le système modélisé n'est composé que d'actions de durées unitaires (afin par exemple de représenter la préemptivité des actions à chaque unité de temps), les réseaux de Petri autonomes avec la règle de tir maximal peuvent remplacer, de façon avantageuse en ce qui concerne la facilité d'implémentation, les RdP temporisés avec la règle de tir maximal, puisque ces derniers nécessitent la prise en compte d'un vecteur de dates de fin des transitions dont le franchissement est en cours.

C’est donc les RdP autonomes avec la règle de tir maximal que nous avons choisi d’utiliser afin de modéliser et d’ordonnancer les systèmes temps réel.

PARTIE II

CONTRIBUTION

III Etude de la Cyclicité des Ordonnancements de Tâches Périodiques

Cette section expose la preuve de cyclicité des séquences d'ordonnancement des systèmes de tâches différées. Pour commencer, elle expose la notion de temps creux, en montrant son rôle dans le problème abordé. Ensuite, la preuve de cyclicité est donnée, sous l'hypothèse que les tâches sont indépendantes, ce qui a pour but de donner au lecteur une idée de la preuve générale. Enfin, le cas des tâches interagissantes est étudié, et la preuve de cyclicité est étendue à ce cas.

Dans le cadre d'une approche en-ligne, l'utilisation de CS ou de CNS peut permettre de se passer de la simulation lorsque les tâches sont indépendantes pour les algorithmes classiques. En effet, la durée de simulation, se basant sur le *ppcm* des périodes des tâches, n'est pas polynomiale. Cependant, dans le cadre d'une étude de performances basée sur une simulation, il est intéressant de simuler un algorithme d'ordonnancement afin d'en vérifier certaines propriétés.

De plus, l'utilité de minimiser la durée de simulation est immédiate pour les approches hors-ligne : en effet quelle doit être la longueur de la séquence à construire ? Comment la minimiser ?

Lorsque les tâches sont simultanées, la durée de simulation est le *ppcm* des périodes P, appelé méta-période (voir Théorème I-2-3).

Par contre, dès lors que des tâches sont différées, à ce jour, le seul résultat portant sur la durée de simulation a été proposé par Leung et Merrill [LM 80][LW 82] (voir Théorème I-2-4) : il énonce que dans le contexte $|1|r_i,C_i,D_i,P_i|$, la durée de simulation suffisante d'un système de tâches ordonnancé par l'algorithme *ED* est de $max_{i=1..n}\{r_i\}+2P$.

C'est pour cette raison, que jusqu'ici aucune étude hors-ligne n'a été étendue aux systèmes de tâches différées : par exemple, Xu et Parnas [XP 90] étudient le contexte non périodique $|1|r_i,C_i,D_i,prec,res|$ et expliquent que l'approche peut se généraliser au contexte périodique $|1|r_i=0,C_i,D_i,P_i,prec,res|$ via un éclatement des tâches périodiques en chacune de leurs occurrences pendant la durée de simulation P. Ainsi, chaque tâche périodique τ_i est remplacée par P/P_i tâches non périodiques.

Il y a donc ici une lacune théorique : comment trouver la durée de simulation minimale nécessaire à la connaissance d'une séquence d'ordonnancement ?

Nous consacrons donc ce chapitre à la minimisation de la durée de simulation d'un système de tâches périodiques différées, de façon indépendante de tout algorithme d'ordonnancement. L'enjeu d'une telle étude est double :

- D'une part, permettre de trouver une durée universelle minimale de simulation d'algorithmes en-ligne.
- D'autre part, permettre aux algorithmes hors-ligne, basés le plus souvent sur des techniques d'énumération de séquences, de se cantonner à une durée d'étude minimale.

III-1 Etude des temps creux acycliques

Afin d'amener progressivement notre étude des ordonnancements des tâches différées, il convient d'expliquer en détail le concept de temps creux acycliques, qui à notre connaissance, n'a jamais été abordé dans la littérature.

Théorème III-1-1 : Soit S un système de tâches simultanées du contexte $|m|r_i=0,C_i,D_i,P_i,$ *prmptnoprmpt*, *prec*, *resRW*$|$, et soit U_s sa charge processeur. Toute séquence d'ordonnancement valide a exactement $P(m-U_s)$ temps creux dans l'intervalle de temps $[0..P[$ où $P=PPCM_{i=1..n}(P_i)$ est la métapériode du système.

Preuve : la preuve découle presque directement de la définition de la charge processeur. En effet, chaque tâche est activée et doit être exécutée en tout P/P_i fois dans $[0..P[$. Les m processeurs doivent donc se partager le traitement de $\frac{PC_i}{P_i}$ quanta de temps correspondant aux occurrences de chaque tâche τ_i. Les processeurs doivent donc dédier en tout $\sum_{i=1}^{n}\frac{PC_i}{P_i}$ quanta de temps au traitement des tâches dans $[0..P[$, c'est à dire par définition de la charge, PU_S. Or leur puissance de traitement est d'un quantum par unité de temps, ce qui signifie qu'ils peuvent traiter mP quanta de temps dans $[0..P[$. Il y a donc exactement $P(m-U_s)$ temps creux dans $[0..P[$.
❑

Ce théorème est assez intuitif puisqu'il découle directement de la définition de la charge processeur. Nous appelons les temps creux considérés des **temps creux cycliques**, puisqu'ils reviennent périodiquement lorsque la séquence de longueur P est répétée indéfiniment.

Cependant, ce théorème ne s'applique pas au cas où des tâches sont différées. Pour illustrer ce fait, étudions l'exemple III-1-1 donné dans le contexte $|1|r_i,C_i,D_i,P_i|$.

---

exemple III-1-1 : soit $S=\{\tau_1<0,1,4,4>,\tau_2<1,3,6,6>,\tau_3<3,1,4,4>\}$ un système de tâches du contexte $|1|r_i,C_i,D_i,P_i|$. La charge du système est $U_S=1$, il ne doit donc pas y avoir de temps creux. La construction de l'ordonnancement créé par *ED* de ce système, donnée sur la figure III-1-1, montre cependant l'existence d'un temps creux.

En effet, la figure III-1-1 montre le début de la séquence σ produite par *ED*. Constatons qu'à l'instant *6*, tout le travail en cours a été traité et que le processeur reste oisif : il y a un temps creux, alors que la charge est de *100%*. De plus, si l'on étudie en détail σ, on s'aperçoit que le système de tâches est dans le même état général à l'instant *19* qu'à l'instant *7*. Cela signifie que la séquence d'ordonnancement donnée par *ED* s'exprime par une montée en charge terminée par le temps creux de l'instant *6* sur l'intervalle, c'est σ prise dans [0..7[que nous dénotons $\sigma_{[0..7[}$, suivie d'une séquence cyclique $\sigma_{[7..19[}$ de longueur *P*. La séquence σ s'écrit donc $\sigma_{[0..7]}\,\sigma_{[7..19[}^{*}$ où l'étoile est l'étoile de Kleene.

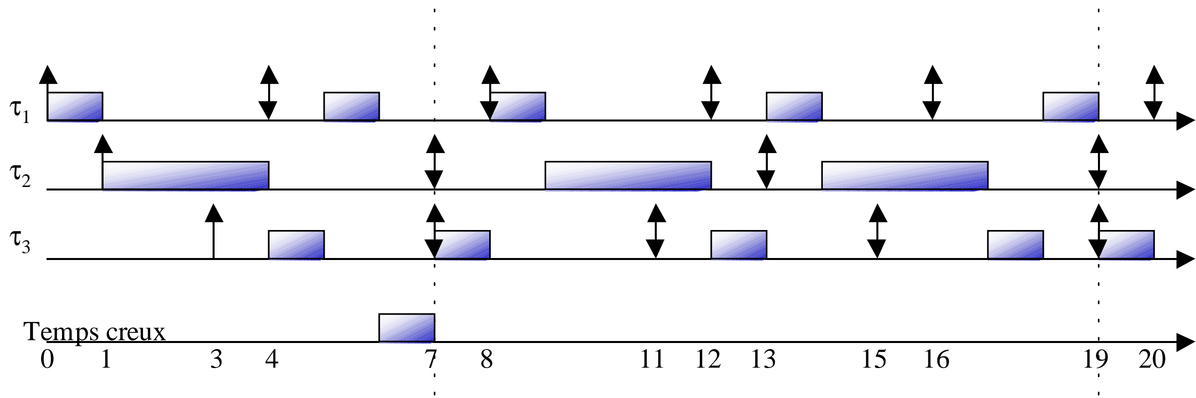


figure III-1-1 : séquence d'ordonnancement de S par ED, un temps creux existe

Cela signifie qu'un temps creux comme celui de l'unité de temps 6 intervient une et une seule fois dans toute la séquence, contrairement aux temps creux cycliques qui apparaissent $P(m-U_S)$ fois toutes les méta-périodes. Nous qualifions ce temps creux de **temps creux acyclique**.

Cet exemple illustre plusieurs faits et soulève plusieurs questions :

1) Certains temps creux peuvent apparaître alors que la charge est maximale, ces temps creux sont-ils dépendants de l'algorithme d'ordonnancement choisi ?
2) La séquence produite sur cet exemple entre dans un cycle de longueur P juste après le temps creux acyclique, est-ce le cas général ?
3) La date d'occurrence des temps creux acycliques dépend-elle de l'algorithme d'ordonnancement choisi, et est elle bornée ?

Une idée de la réponse à la première question est donnée par la construction du **diagramme de charge processeur**, du système S, qui est donné sur la figure III-1-2.

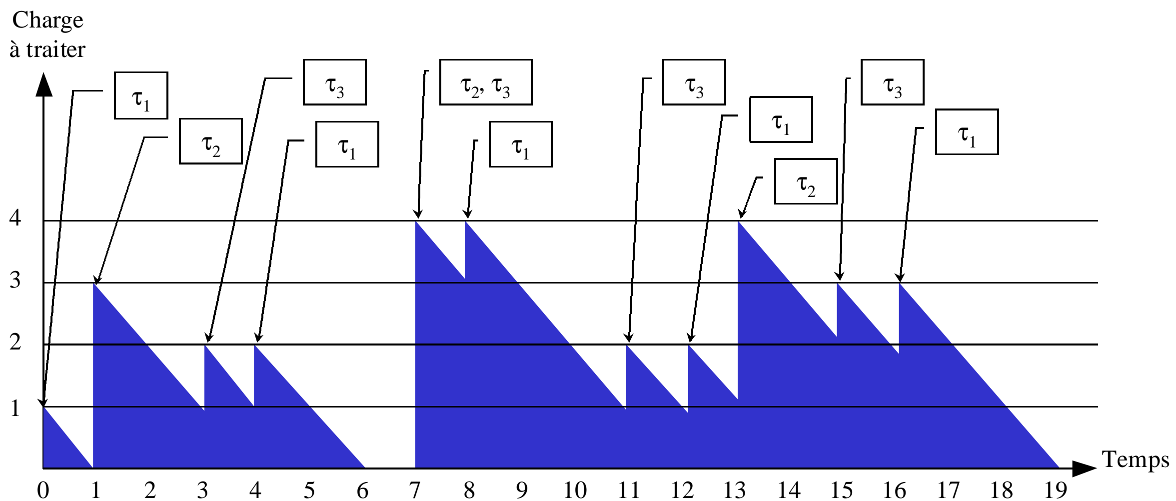


figure III-1-2 : charge induite par les requêtes au cours du temps

A chaque réveil d'une tâche correspond une requête de temps processeur. Si l'algorithme d'ordonnancement est conservatif, c'est à dire qu'il ne reste pas inactif lorsque des tâches peuvent être traitées, alors le système S est tel qu'il y a un temps creux à l'instant *6*. En effet,

la figure montre qu'à cet endroit plus aucune tâche ne requiert le processeur, et ce quel que soit l'algorithme conservatif choisi.

Si les tâches sont indépendantes, un tel diagramme permet de trouver tous les temps creux du système. Si la charge des tâches est de *100%*, tous les temps creux sont des temps creux acycliques. Si ce n'est pas le cas, il nous faut trouver un moyen de les dissocier. Afin d'écarter provisoirement cet obstacle, étudions en détail le cas des systèmes de tâches différées de charge *100%.*

Nous nous plaçons pour notre preuve dans un contexte monoprocesseur. Nous avons vu sur l'exemple III-1-1 que les temps creux acycliques semblaient avoir une importance sur la cyclicité des séquences d'ordonnancement.

La première étape de notre étude consiste à démontrer qu'aucun temps creux acyclique ne peut se trouver dans la partie cyclique d'un ordonnancement.

Lemme III-1-1 : Soit S un système de tâches ordonnancé dans le contexte $|1|r_i,C_i,D_i,P_i,prmptnoprmpt,prec,resRW|$, avec U_S=1. Le nombre de temps creux présents dans une séquence d'ordonnancement valide de S est borné.

<u>Preuve :</u> Voir annexe I-4.

❏

Ce lemme permet la démonstration immédiate du lemme suivant :

Lemme III-1-2 : Soit S un système de tâches ordonnancé sur 1 processeur dans le contexte $|1|r_i,C_i,D_i,P_i,prmptnoprmpt,prec,resRW|$, avec U_S=1. Soit σ_S une séquence d'ordonnancement valide de S s'écrivant sous la forme $\sigma_S=\sigma_m\sigma_c^*$ où l'étoile est l'étoile de Kleene. Aucun temps creux ne peut être présent dans σ_c.

<u>Preuve :</u> La preuve découle du Lemme III-1-1. puisque si un temps creux est présent dans σ_c, alors le nombre de temps creux n'est pas borné dans σ.

❏

Ce lemme montre que si un ordonnancement s'écrit sous la forme montée en charge, partie cyclique, alors la partie cyclique ne peut pas englober de temps creux. La partie cyclique ne peut donc commencer qu'après le dernier temps creux, qui, puisque nous nous plaçons dans le cadre de systèmes de tâches chargés à 100%, est forcément un temps creux acyclique.

La partie cyclique ne pouvant en aucun cas commencer avant le dernier temps creux acyclique, il est intéressant de savoir si sa date d'occurrence est bornée dans le temps.

III-1.1 Cas des systèmes de tâches indépendantes

Afin de simplifier notre raisonnement, nous commençons par nous intéresser au contexte $|1|r_i,C_i,D_i,P_i|$, puis nous étendrons les preuves au contexte général $|1|r_i,C_i,D_i,P_i, prmptnoprmpt, prec,resRW|$.

Nous montrons dans cette partie que la date d'occurrence du dernier temps creux acyclique est bornée. Dans ce but, il nous faut introduire quelques notations portant sur la charge des tâches.

Définition III-1-1 : La **requête instantanée** d'une tâche τ_i est définie par $\boldsymbol{C_{req}}: \mathbb{N} \times \{1..n\} \rightarrow \mathbb{N}$, $\boldsymbol{C_{req}}(t,i)=\begin{cases} C_i \text{ si } t \equiv r_i\,[P_i] \\ 0 \text{ sinon} \end{cases}$.

La requête instantanée d'un système est $\boldsymbol{C_{req}}(t)=\sum_{i=1}^{n} C_{req}(t,i)$

La requête instantanée est la charge correspondant à l'activation des tâches. On peut remarquer que $C_{req}(t)$ est périodique de période P pour $t>r$.

Enfin, nous définissons récursivement la charge instantanée restant à traiter.

Définition III-1-2 : La **charge instantanée**, ou **charge restante**, à l'instant t est définie par $C_{rest}: \mathbb{N} \rightarrow \mathbb{N}$, avec :

$C_{rest}(0)=C_{req}(0)$

$C_{rest}(t)= C_{req}(t) + max\{0, C_{rest}(t-1) -1\}$.

Cette définition est la définition la plus générale permettant de caractériser un algorithme d'ordonnancement conservatif traitant des systèmes de tâches indépendantes de charge *100%* : si la charge restant à traiter est non nulle, alors le processeur travaille.

Il est à noter que, comme l'exemple III-1-2 l'indique, ce comportement ne caractérise pas les ordonnancements multiprocesseurs.

exemple III-1-2 : soit un système de tâches $S=\{\tau_1<0,4,4,4>,\tau_2<2,4,4,4>\}$, de charge $U_S=2$, c'est à dire *100%*, ordonnancé dans le contexte $|2|r_i,C_i,D_i,P_i|$. La figure III-1-3 montre une séquence d'ordonnancement valide de ce système : deux temps creux apparaissent initialement sur le deuxième processeur, car une tâche ne peut être exécutée simultanément sur plusieurs processeurs. Cependant, le diagramme des charges du système, représenté sur la figure III-1-4, montre que la charge restant à traiter n'est jamais nulle.

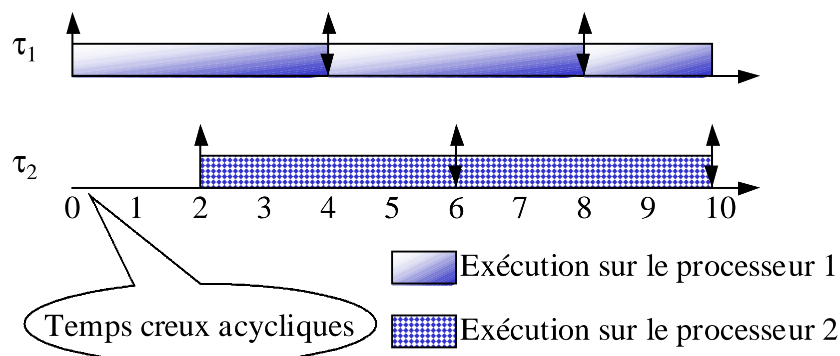


figure III-1-3 : temps creux acycliques pendant un ordonnancement multiprocesseur

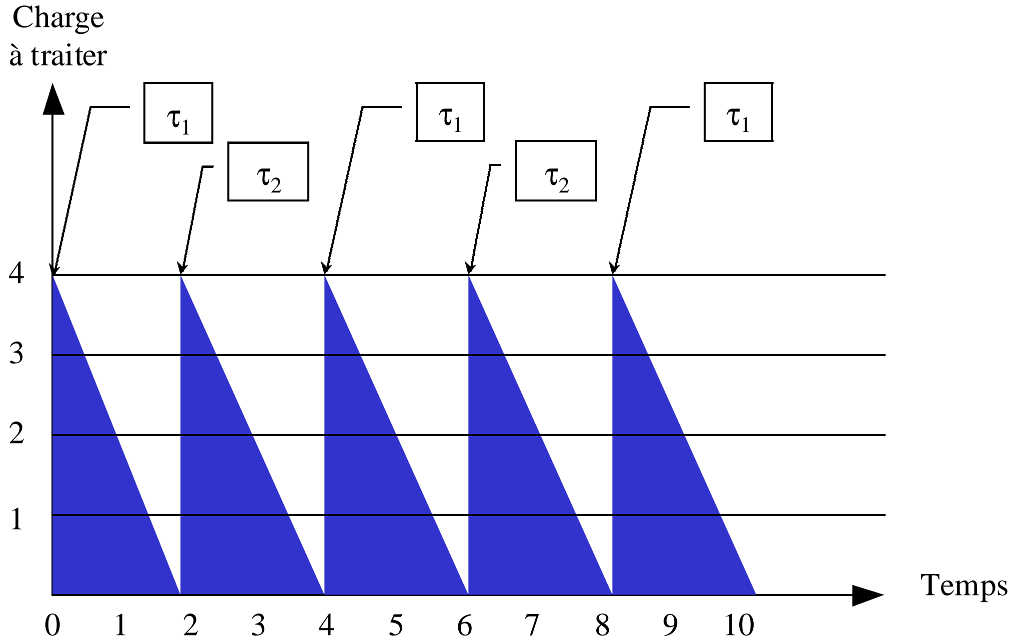


figure III-1-4 : diagramme de charge correspondant

Il faut donc souligner que la preuve que nous présentons ne s'étend pas naturellement aux systèmes multiprocesseur.

Dans un premier temps, nous montrons que la date d'occurrence d'un temps creux acyclique est bornée par $r+P$, en montrant que pour tout entier Δ positif ou nul, il n'y a jamais de temps creux à une date $r+P+\Delta$.

Pour commencer, nous avons besoin de deux lemmes instrumentaux.

Lemme III-1-3 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i|$ tel que $U_S=1$. Après la date $r=max_{i=1..n}\{r_i\}$, la somme des requêtes instantanées est de P unités de temps processeur à allouer toutes les P unités de temps. C'est à dire $\sum_{i=0}^{P-1} C_{req}(r+i) = P$.

<u>Preuve :</u> La preuve se base sur la définition de la charge. Chaque tâche τ_i a exactement P/P_i requêtes toutes les P unités de temps, induisant chacune une demande de C_i unités de temps processeur. Puisque l'on se place après la date r, $\sum_{i=0}^{P-1} C_{req}(r+i) = \sum_{i=1}^{n} \frac{PC_i}{P_i}$, ce qui, puisque la charge $\sum_{i=1}^{n} \frac{C_i}{P_i}$=1, donne directement le résultat du lemme.

❑

Lemme III-1-4 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i|$ tel que $U_S=1$. La charge restant à traiter à un instant $t+P$ ne peut pas être inférieure à celle qui restait à traiter à la date t pour $t \geq r$. C'est à dire :

$\forall \Delta \geq 0, C_{rest}(r+P+\Delta) \geq C_{rest}(r+\Delta)$

Preuve : voir annexe I-4.

❑

III-1.1.1 La date d'occurrence du dernier temps creux acyclique est bornée

Afin de montrer la bornitude de la date d'occurrence des temps creux acycliques, deux lemmes sont nécessaires :

Lemme III-1-5 :Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i|$ tel que $U_S=1$. Si il y a un temps creux à un instant $t\geq r$, alors il n'y en a pas à la date $t+P$. C'est à dire : $\forall\Delta\geq 0$, $C_{rest}(r+\Delta)=0 \Rightarrow C_{rest}(r+P+\Delta)>0$

Preuve : La preuve formelle est similaire à celle du Lemme III-1-4 et n'est pas détaillée ici. Elle est basée sur le fait suivant : puisque la charge restante est nulle à l'instant $r+\Delta$, il y a un temps creux. Or les requêtes des tâches dans l'intervalle $[r+\Delta..\ r+P+\Delta[$ nécessitent P unités de temps de traitement d'après la définition de la charge maximale. Le processeur étant resté oisif pendant au moins une unité de temps (la charge étant nulle à l'instant $r+\Delta$), il n'aura pu au plus traiter que $P-1$ quanta de temps. Il reste donc au moins une unité de temps à traiter à la date $r+P+\Delta$.

❑

Le deuxième lemme nécessaire à la démonstration de la première proposition est le suivant :

Lemme III-1-6 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i|$ tel que $U_S=1$. Si il n'y a pas de temps creux à un instant $t\geq r$, alors il n'y en a pas à la date $t+P$. C'est à dire : $\forall\Delta\geq 0$, $C_{rest}(r+\Delta)>0 \Rightarrow C_{rest}(r+P+\Delta)>0$.

Preuve : découle directement du Lemme III-1-4.

❑

Proposition III-1-1: Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i|$ tel que $U_S=1$. Aucun temps creux ne peut apparaître à partir de la date $r+P$. C'est à dire $\forall\Delta\geq 0$, $C_{rest}(r+P+\Delta)>0$.

Preuve : D'après le Lemme III-1-5 et le Lemme III-1-6, $\forall\Delta\geq 0$, $C_{rest}(r+P+\Delta)>0$ quelle que soit la valeur de $C_{rest}(r+\Delta)$.

❑

Les temps creux acycliques ne peuvent donc intervenir qu'avant la date $r+P$, et ce quel que soit l'algorithme d'ordonnancement conservatif choisi.

III-1.1.2 L'état d'un système est identique après le dernier temps creux acyclique et une méta-période plus tard

Nous montrons dans cette section qu'après le dernier temps creux acyclique, le système est dans le même état toutes les P unités de temps.

Définition III-1-3 : Nous notons $\mathbf{t_c}$ la date d'occurrence du dernier temps creux acyclique. Par convention, nous choisissons t_c=-1 si il n'y a pas de temps creux acyclique dans une séquence.

La proposition suivante montre que l'état global du système de tâches est forcément le même aux instants t_c+1 et P unités de temps plus tard.

Proposition III-1-2 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i|$ tel que U_S=1. Toute séquence d'ordonnancement conservative de S valide sur l'intervalle $[0..t_c+P+1[$ laisse le système dans le même état à l'instant t_c+P+1 qu'à l'instant t_c+1.

Preuve : voir annexe I-4.

❑

Enfin, les deux propositions précédentes permettent de prouver le théorème suivant :

Théorème III-1-2 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i|$ tel que U_S=1, et soit σ une séquence conservative valide sur l'intervalle $[0..t_c+P+1[$:

- la séquence $\sigma_{[0..t_c+1[}\sigma^*_{[t_c+1..t_c+P+1[}$, où $\sigma_{[a..b[}$ est la restriction de σ à l'intervalle $[a..b[$, est valide ;
- le dernier temps creux acyclique a lieu à une date $t_c<r+P$

Preuve : $t_c<r+P$ est le résultat de la Proposition III-1-1, et la périodicité résulte de la Proposition III-1-2.

❑

La Proposition III-1-2 implique aussi le fait que si l'algorithme choisi est déterministe (i.e. face à la même configuration de tâches, il prend toujours la même décision), la validité d'une séquence sur l'intervalle $[0..t_c+P+1[$ est une condition nécessaire et suffisante d'ordonnançabilité puisque la séquence infinie produite correspond à $\sigma_{[0..t_c+1[}\sigma^*_{[t_c+1..t_c+P+1[}$.

Remarque : certains systèmes de tâches différées n'ont pas de temps creux acyclique. Il résulte que t_c varie, suivant les systèmes, entre -1 et $r+P$-1, ce qui implique que la durée nécessaire à l'obtention du régime permanent d'un système varie entre P et $r+2P$.

III-1.2 Cas des systèmes de tâches quelconques

Bien qu'intéressant, ce résultat n'est pas suffisant car le contexte pris en compte (pas de précédence, pas de section critique) n'est pas réaliste. Par ailleurs, aucune approche hors-ligne ne s'intéresse à un tel contexte puisqu'*ED* y est optimal.

Il faut donc nous intéresser au contexte $|1|r_i,C_i,D_i,P_i$, *prmptnoprmpt*, *prec*, *resRW*|. Nous pouvons nous limiter à l'étude du contexte $|1|r_i,C_i,D_i,P_i,$*prec*,*resRW*| puisqu'il est possible de simuler des parties non préemptibles de tâches par l'utilisation de ressources critiques.

L'intégration des ressources critiques, qu'elles soient exclusives ou en lecture/écriture, n'entraîne pas de modification de notre preuve, étant donné que des temps creux n'ont lieu que lorsqu'aucune tâche ne peut travailler. Par contre, l'intégration des contraintes de précédence complique un peu les choses. Nous posons l'hypothèse, très réaliste, suivante : **une tâche ne peut pas attendre de message à l'intérieur d'une section critique**.

Sous cette hypothèse, permettant de séparer les blocages dus aux attentes de messages des blocages dus à l'attente de ressources, il y a un temps creux si et seulement si :

1. la charge restant à traiter est nulle, comme pour le cas des tâches indépendantes,
2. ou bien toutes les tâches réveillées sont bloquées en attente d'une ressource critique. Dans ce cas, le système est bloqué : on parle d'interblocage, et la séquence d'ordonnancement considérée ne peut pas être valide. Nous ignorons donc ce cas,
3. ou bien toutes les tâches réveillées sont bloquées en attente d'un message.

Notre hypothèse de travail permet d'éliminer le cas où une tâche attend une ressource qui est détenue par une tâche qui attend un message d'une tâche.

Par contre, elle n'exclut pas le cas 3. Ce cas a lieu si toutes les tâches actives attendent un message d'une autre tâche, qui elle-même, en attend un d'une autre tâche, et ainsi de suite jusqu'à une tâche qui attend un message d'une tâche qui n'est pas réveillée. On parle alors de chaîne d'attente.

III-1.2.1 Définition des chaînes d'attente

Nous avons vu en section I-2.2.2 que les tâches communicantes devaient être contraintes de communiquer à la même fréquence (i.e. la fréquence d'émission de la tâche émettrice est la même que la fréquence de réception de la tâche réceptrice). Nous prenons cette hypothèse de départ. Elle est réaliste puisque si la fréquence de réception est plus élevée que celle d'émission, le système de tâches n'est pas ordonnançable, et si c'est l'inverse, alors il y a des écrasements de messages : on ne peut plus alors parler de contraintes de précédence. Dans la suite, nous utilisons des boîtes aux lettres pour modéliser les communications, mais nous ne considérons que les communications 1-1, c'est à dire que pour chaque couple producteur-consommateur de messages, il existe une boîte aux lettres $bal_{producteur,consommateur}$. Le modèle de communication considéré est représenté sur la figure III-1-5.

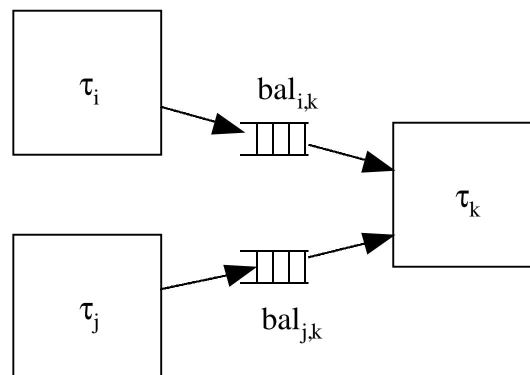


figure III-1-5 : modèle de communications considéré

Donc, soient deux tâches τ_i et τ_j communiquant l'une avec l'autre via une boîte aux lettres $bal_{i,j}$. Soit $\mathbf{db_{j,i}}$ le nombre d'instructions *Déposer_BAL*($bal_{j,i}$,δ) contenues dans le corps de τ_j et soit $\mathbf{rb_{i,j}}$ le nombre d'instructions *Retirer_BAL*($bal_{j,i}$, δ) contenues dans le corps de τ_i. Alors par hypothèse $\frac{db_{j,i}}{P_i} = \frac{rb_{i,j}}{P_j}$.

Afin de faciliter les preuves à venir, et étant donné que $db_{j,i}$ instructions *Déposer_BAL*($bal_{j,i}$,δ) peuvent exister dans le corps d'une tâche τ_j, on les indice dans leur ordre d'apparition *Déposer_BAL*($bal_{j,i}$,δ)$_1$, *Déposer_BAL*($bal_{j,i}$,δ)$_2$, …, *Déposer_BAL*($bal_{j,i}$,δ)$_{db_{j,i}}$. On fait de même pour les $rb_{i,j}$ instructions correspondantes dans le corps de τ_i qui sont notées *Retirer_BAL*($bal_{j,i}$, δ)$_1$, *Retirer_BAL*($bal_{j,i}$, δ)$_2$, …, *Retirer_BAL*($bal_{j,i}$, δ)$_{rb_{i,j}}$.

Enfin, comme sur la figure III-1-6, on note $CD_{j,i,k}$ la charge de τ_j précédant l'instruction *Déposer_BAL*($bal_{j,i}$,δ)$_k$ et de façon symétrique $CR_{i,j,k}$ la charge de τ_i précédant l'instruction *Retirer_BAL*($bal_{j,i}$, δ)$_k$.

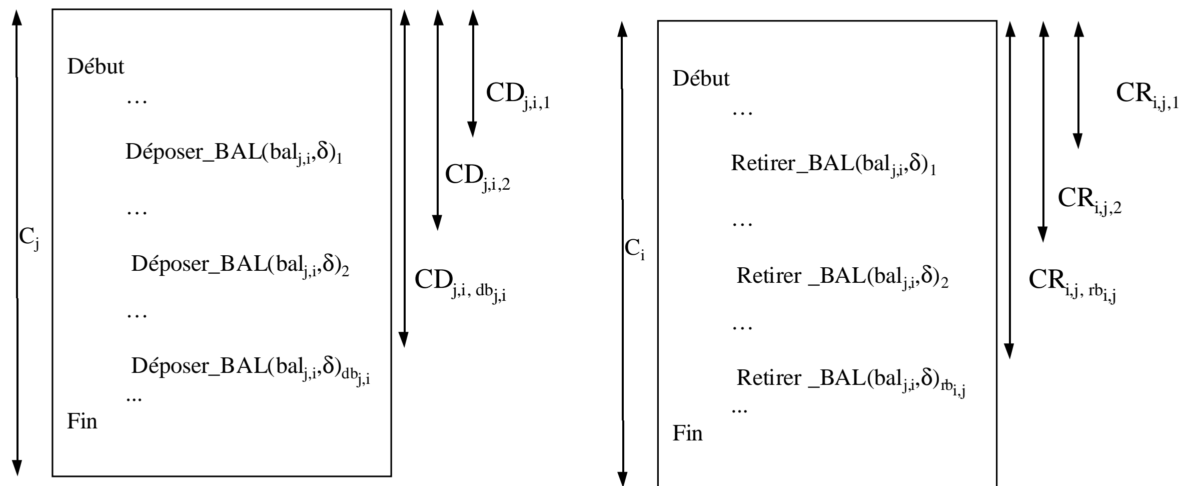


figure III-1-6 : notations utilisées pour dénoter la charge précédant les attentes et envois de messages

Sous l'hypothèse de mêmes fréquences d'émission et de réception de toute paire de tâches communicantes, une attente intervient lorsqu'une tâche émettrice est tardive par rapport à la réceptrice associée, comme c'est explicité sur la figure III-1-7.

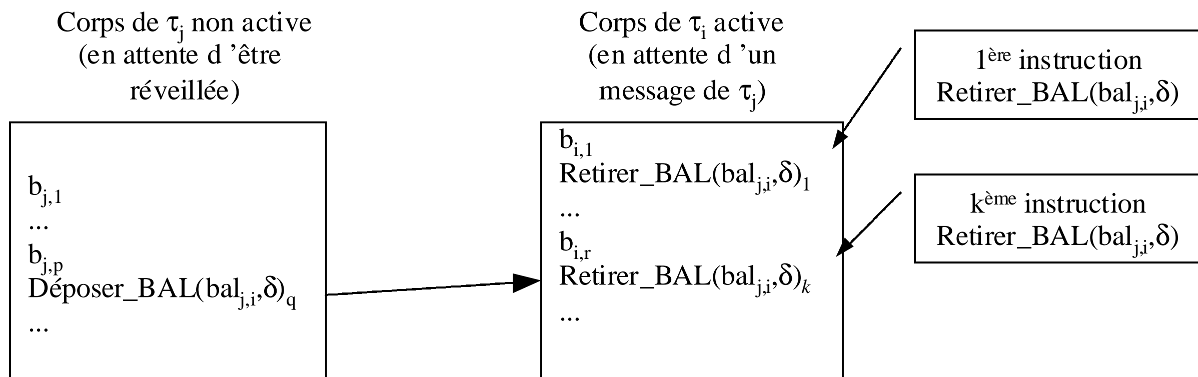


figure III-1-7 : une attente directe

Sur cette figure, la tâche τ_i est en attente au niveau de l'instruction $Retirer_BAL(bal_{j,i},\delta)_k$: si τ_i est la seule tâche active et qu'elle se trouve au niveau de cette instruction, alors il y a un temps creux alors que la charge restant à traiter est non nulle. C'est le cas si $C_{rest}(t)=C_i-CR_{i,j,k}$.

Les définitions suivantes nous permettent de formaliser la notion d'attente.

Définition III-1-4 : Nous appelons **attente directe** l'attente par une tâche d'un message venant de l'occurrence d'une autre tâche non activée à l'instant t. Une attente directe est donnée sous la forme d'un triplet $\{\tau_i, k, \tau_j\}$, qui signifie que l'instruction $Retirer_BAL(bal_{j,i}, \delta)_k$ du corps de la tâche τ_i, si elle a lieu à l'instant t, met obligatoirement la tâche τ_i en attente car le message attendu ne sera fourni que par une requête de la tâche émettrice τ_j qui n'est pas encore réveillée.

L'ensemble des attentes directes au temps t, noté $\boldsymbol{AD(t)}$, est donc défini par :

$$AD(t)=\left\{\{\tau_i, k, \tau_j\} / k \in [1..rb_{i,j}], \left(k + rb_{i,j}\left\lfloor\frac{t-r_i}{P_i}\right\rfloor > db_{j,i}\left\lceil\frac{t-r_j}{P_j}\right\rceil\right) \wedge (t \geq r_i)\right\}$$

Cette définition n'est pas d'un abord facile, nous explicitons donc le concept d'attente directe : soit $\{\tau_i, k, \tau_j\}$ une attente directe à l'instant t. L'entier $k\in[1,..,rb_{i,j}]$ est le numéro de l'instruction $retirer_BAL(bal_{j,i})$ qui ne peut en aucun cas être satisfaite avant le prochain réveil de τ_j. Avant la date t, les requêtes terminées de la tâche τ_i ont consommé exactement $rb_{i,j}\left\lfloor\frac{t-r_i}{P_i}\right\rfloor$ messages de τ_j, le terme $k + rb_{i,j}\left\lfloor\frac{t-r_i}{P_i}\right\rfloor$ représente donc le nombre de messages de τ_j que peut attendre τ_i dans son instance courante à la date t, avec k variant de 1 au nombre de messages de τ_j que peut attendre chaque occurrence de τ_i. L'occurrence courante de τ_i à la date t devra consommer 1 message de τ_j, puis un autre, et encore un autre, etc… exactement $rb_{i,j}$ fois. Cependant, τ_j n'a peut être pas pu fournir suffisamment de messages à l'instant t, puisqu'à cette date, τ_j a pu émettre au plus $db_{j,i}\left\lceil\frac{t-r_j}{P_j}\right\rceil$ messages. Si c'est le cas, alors la tâche τ_i ne pourra en aucun cas être exécutée au delà de l'attente de message correspondante.

Etudions à nouveau la figure III-1-7 : si il y avait d'autres instructions $Retirer_BAL(bal_{j,i},\delta)$ après la $k^{ème}$, celles-ci feraient aussi partie de l'ensemble $AD(t)$. C'est à dire que si un triplet $\{\tau_i, k, \tau_j\}\in AD(t)$, alors $\forall g\in\{k+1,.., rb_{i,j}\}$, $\{\tau_i, g, \tau_j\}\in AD(t)$. De plus, une tâche τ_i peut attendre des messages de différentes tâches, donc il est possible de trouver la tâche τ_i plusieurs fois avec des tâches différentes dans $AD(t)$.

Les blocages sont transitifs, puisqu'une tâche τ_i apparaissant dans $AD(t)$, bien qu'étant active, peut bloquer d'autres tâches de la même façon, si certaines de ses émissions sont placées après l'attente bloquante. On parle alors d'**attente**, notée $\boldsymbol{A(t)}$, qui est défini par :

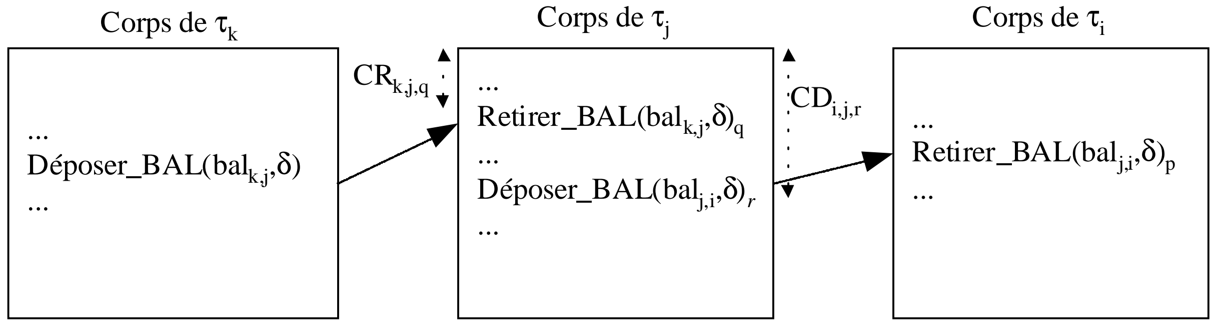
$$A(t)=AD(t) \cup$$

$$\left\{\{\tau_i, p, \tau_j\} / t \geq r_i \wedge \exists\{\tau_j, q, \tau_k\} \in A(t), \exists r \in \{1..db_{j,i}\}, p + rb_{i,j}\left\lfloor\frac{t-r_i}{P_i}\right\rfloor \geq db_{j,i}\left\lfloor\frac{t-r_j}{P_j}\right\rfloor / CD_{j,i,r} > CR_{j,k,q}\right\}$$

En effet, la tâche est en attente au niveau d'une instruction *Retirer_BAL*($bal_{j,i}$,δ)$_p$ si et seulement :

- c'est une attente directe (i.e. elle est dans $A(t)$),
- la tâche τ_j est en attente d'un message provenant d'une occurrence d'une tâche τ_k qui peut être active mais qui est en attente avant l'instruction *Déposer_BAL*($bal_{j,i}$,δ)$_p$ d'émission du message qui permettrait à τ_j d'envoyer le message attendu par τ_i (ce qui est le cas si $CD_{j,i,r}>CR_{j,k,q}$). C'est une **attente indirecte** illustrée sur la figure III-1-8.

Nous nous intéressons aux blocages dus aux messages car des temps creux apparaîtront si et seulement si la charge restant à traiter à un instant donné fait intégralement partie de tâches en attente.



$$\{\tau_j, q, \tau_k\} \in A(t) \wedge p + rb_{i,j}\left\lfloor \frac{t - r_i}{P_i} \right\rfloor \geq db_{j,i}\left\lfloor \frac{t - r_j}{P_j} \right\rfloor \wedge CD_{j,i,r} > CR_{k,j,q} \Rightarrow \{\tau_i, r, \tau_j\} \in A(t)$$

figure III-1-8 : illustration d'une attente indirecte

Remarquons que l'ensemble $A(t)$ est le point fixe de la suite suivante :
$A_0(t)=AD(t)$, et $\forall n \in \mathbb{N}$, $A_n(t)=A_{n-1}(t) \cup$

$$\left\{\{\tau_i, p, \tau_j\} / t \geq r_i \wedge \exists\{\tau_j, q, \tau_k\} \in A_{n-1}(t), \exists r \in \{1..db_{j,i}\}, p + rb_{i,j}\left\lfloor \frac{t - r_i}{P_i} \right\rfloor \geq db_{j,i}\left\lfloor \frac{t - r_j}{P_j} \right\rfloor / CD_{j,i,r} > CR_{j,k,q}\right\}$$

III-1.2.2 Périodicité des chaînes d'attente

Nous avons vu qu'un temps creux intervenait si à un instant t toute la charge restant à traiter était située après des attentes de messages ne pouvant être satisfaites. Afin de vérifier que le résultat de cyclicité des ordonnancements reste valide dans le cas général, nous allons étudier la cyclicité des chaînes d'attente.

Lemme III-1-7 : Les attentes qui se trouvent après la date P englobent les attentes qui se trouvent une méta-période plus tôt. C'est à dire $\forall\Delta\geq 0$, $A(\Delta)\subseteq A(P+\Delta)$.

Preuve : voir annexe I-4.
❑

Ce lemme montre qu'il y a conservation des chaînes d'attente tout au long des méta-périodes, ce qui est naturel intuitivement puisque les attentes sont données par la structure des

tâches communicantes, qui ont la même fréquence. Le lemme nous servira à montrer la périodicité des attentes.

Lemme III-1-8 : $\forall\Delta\geq 0$, $A(r+P+\Delta)\subseteq A(r+\Delta)$.

Preuve : la preuve est la même que celle du lemme précédent.
❑

Proposition III-1-3 : Les attentes sont périodiques de période P à partir du moment où toutes les tâches ont été activées. C'est à dire $\forall\Delta\geq 0$, $A(r+\Delta)=A(r+P+\Delta)$.

Preuve : découle du Lemme III-1-7 et du Lemme III-1-8.
❑

Les attentes sont donc périodiques à partir de r. Cela nous permet de reprendre la preuve de cyclicité des ordonnancements de tâches indépendantes afin de l'étendre au cas des tâches communicantes.

III-1.2.3 Cyclicité des ordonnancements dans le cas général

La définition de requête instantanée est la même que dans le cas où les tâches sont indépendantes. Cependant, la charge instantanée doit être redéfinie, puisque les temps creux apparaissent lorsque la charge restante correspond à la charge en attente, c'est à dire la charge restant à traiter après les points d'attente des tâches de $A(t)$.

Définition III-1-5 : La **charge en attente** d'une tâche τ_i est définie par $C_{att}: I\!N \times \{1..n\} \rightarrow I\!N$,

$C_{att}(t,i)=max_{k,j}\{C_i-CR_{i,j,k}, \{\tau_i,k,\tau_j\}\in A(t)\}$.

La requête instantanée d'un système est $C_{att}(t)=\sum_{i=1}^{n} C_{att}(t,i)$

La charge en attente (voir figure III-1-6) correspond à la charge qui ne peut pas être traitée à la date t à cause des attentes de messages. Lorsque la charge restante est égale à la charge en attente, toutes les tâches actives sont en attente, et il y a un temps creux. D'où la modification de la définition de la charge instantanée.

Définition III-1-6 : La **charge instantanée**, ou **charge restante**, à l'instant t est définie par $C_{rest}: I\!N \rightarrow I\!N$, avec :

$C_{rest}(0)=C_{req}(0)$

$C_{rest}(t)= C_{req}(t)+ C_{rest}(t-1) -\gamma(t-1)$ avec $\gamma: I\!N \rightarrow \{0,1\}$ définie par :

$$\gamma(t)=\begin{cases} 1 \text{ si } C_{rest}(t) > C_{att}(t) \\ 0 \text{ sinon} \end{cases}.$$

La fonction γ est la fonction caractéristique des ordonnanceurs conservatifs : si il existe des tâches qui ne sont pas en attente d'un message, il en exécute une, sinon, il reste oisif.

Nous reprenons la preuve développée pour les tâches indépendantes que nous adaptons à la nouvelle définition de la charge restante. Cependant, avant cela, nous donnons un lemme instrumental traitant de la charge en attente.

Lemme III-1-9 : $\forall\Delta\geq 0$, $C_{att}(P+\Delta)\geq C_{att}(\Delta)$ et $C_{att}(r+\Delta)=C_{att}(r+P+\Delta)$.

Preuve : l'inégalité découle du Lemme III-1-7, et l'égalité du Lemme III-1-8.
❑

Lemme III-1-10 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i,prec,resRW|$ tel que U_S=1. Après la date r, la somme des charges de requêtes est de P unités de temps à traiter toutes les P unités de temps. C'est à dire $\sum_{i=1}^{P} C_{req}(r+i)=P$.

Preuve : La preuve, basée sur la définition de la charge d'un système, n'est pas modifiée par rapport au Lemme III-1-3.
❑

Lemme III-1-11 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i,prec,resRW|$ tel que U_S=1. La charge restant à traiter à un instant $t+P$ ne peut pas être inférieure à celle qui restait à traiter à la date t pour $t\geq r$. C'est à dire :
$\forall\Delta\geq 0$, $C_{rest}(r+P+\Delta)\geq C_{rest}(r+\Delta)$

Preuve : voir annexe I-4.
❑

III-1.2.3.a La date d'occurrence du dernier temps creux acyclique est bornée

Lemme III-1-12 :Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i, prec,resRW|$ tel que U_S=1. Si il y a un temps creux à un instant $t\geq r$, alors il n'y en a pas à la date $t+P$. C'est à dire :
$\forall\Delta\geq 0$, $\gamma(r+\Delta)=0\Rightarrow\gamma(r+P+\Delta)=1$

Preuve : voir annexe I-4.
❑

Lemme III-1-13 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i, prec,resRW|$ tel que U_S=1. Si il n'y a pas de temps creux à un instant $t\geq r$, alors il n'y en a pas à la date $t+P$. C'est à dire :
$\forall\Delta\geq 0$, $\gamma(r+\Delta)=1\Rightarrow\gamma(r+P+\Delta)=1$.

Preuve : Par définition, $\gamma(r+\Delta)=1\Leftrightarrow C_{rest}(r+\Delta)>C_{att}(r+\Delta)$ **(a)**

Or d'après le Lemme III-1-11, $C_{rest}(r+P+\Delta)\geq C_{rest}(r+\Delta)$ **(b)**

De plus, d'après le Lemme III-1-9, $C_{att}(r+P+\Delta)=C_{att}(r+\Delta)$ **(c)**

D'après (a), (b) et (c) on obtient $C_{rest}(r+P+\Delta)>C_{att}(r+P+\Delta)$.

❑

Proposition III-1-4: Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i, prec,resRW|$ tel que $U_S=1$. Aucun temps creux ne peut apparaître à partir de la date $r+P$. C'est à dire $\forall\Delta\geq 0$, $\gamma(r+P+\Delta)=1$.

Preuve : D'après le Lemme III-1-12 et le Lemme III-1-13, $\forall\Delta\geq 0$, $\gamma(r+P+\Delta)=1$ quelle que soit la valeur de $\gamma(r+\Delta)$.

❑

III-1.2.3.b L'état d'un système est identique après le dernier temps creux acyclique et une méta-période plus tard

Il ne reste plus, pour obtenir le théorème général, qu'à montrer que l'état global du système de tâches est forcément le même aux instants t_c+1 et P unités de temps plus tard.

Proposition III-1-5 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i, prec,resRW|$ tel que $U_S=1$. Toute séquence d'ordonnancement conservative valide de S sur l'intervalle $[0..t_c+P+1[$ laisse le système dans le même état à l'instant t_c+P+1 qu'à t_c+1.

Preuve : La preuve donnée en annexe I-4.

❑

La démonstration de cette proposition demande le même raisonnement que dans le cas des tâches indépendantes. La différence réside dans le fait suivant : lorsque les tâches sont indépendantes, la charge à traiter à l'instant suivant le dernier temps creux acyclique est donnée uniquement par les réactivation des tâches, et il en est de même une méta-période plus tard car les réactivations sont les mêmes. Dans le cas des tâches communicantes, la charge à traiter à l'instant suivant le dernier temps creux acyclique est donnée par les réactivations à cette date plus la charge en attente que le processeur n'a pas pu traiter à l'instant t_c. Une méta-période plus tard, la charge est la même : elle est donnée par les mêmes réactivations, plus le résidu qui subsistait à l'instant t_c. Il nous faut vérifier que ce résidu correspond aux mêmes tâches à l'instant t_c+P+1 qu'à l'instant t_c+1. C'est la périodicité des chaînes d'attente qui nous permet d'affirmer ce fait.

Enfin, les deux propositions précédentes permettent de prouver le théorème suivant :

Théorème III-1-3 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i, prec,resRW|$ tel que $U_S=1$, et soit σ une séquence conservative valide sur l'intervalle $[0..t_c+P+1[$:

- la séquence $\sigma_{[0..t_c+1[}\sigma^*_{[t_c+1..t_c+P+1[}$, où $\sigma_{[a..b[}$ est la restriction de σ à l'intervalle $[a..b[$, est valide ;
- le dernier temps creux acyclique a lieu à une date $t_c<r+P$

Preuve : $t_c<r+P$ est le résultat de la Proposition III-1-4, et la périodicité résulte de la Proposition III-1-5.

❑

exemple III-1-3 : Soit S={τ_1<0,1,4,4>, τ_2<1,1,4,4>, τ_3<2,1,4,4>, τ_4<0,1,4,4>} un système de tâches du contexte $|1|r_i,C_i,D_i,P_i,prec|$, avec :

```
Tâche τ1 est
-- Attend un message
-- de τ2, dure une unité de
-- temps puis émet un
-- message à τ4
r1 = 0 ;
P1 = 4 ;
D1 = 4 ;
Début
  Retirer_BAL(MB2,1)
  b1,1=1 ;
  Déposer_BAL(MB1,4) ;
Fin ;

Tâche τ2 est
-- Dure une unité de
-- temps puis émet un
-- message à τ1
r2 = 1 ;
P2 = 4 ;
D2 = 4 ;
Début
  b2,1=1 ;
  Déposer_BAL(MB2,1)
Fin

Tâche τ3 est
-- Dure une unité de temps
r3 = 2 ;
P3 = 4 ;
D3 = 4 ;
Début
  b3,1=1 ;
Fin ;

Tâche τ4 est
-- Attend un message de τ1 puis
-- dure une unité de temps
r4 = 0 ;
P4 = 4 ;
D4 = 4 ;
Début
  Retirer_BAL(MB1,4)
  b4,1=1 ;
Fin ;
```

Le calcul des chaînes d'attente donne $C_{att}(0)= C_{att}(4)=2$, et C_{att} est nul pour les autres dates comprises entre 0 et 6. Le diagramme de charge donné sur la figure III-1-9 montre qu'il y a un temps creux à l'instant 0, d'où $t_c=0$. Toute séquence conservative σ valide sur [0..5[est valide : la séquence infinie est alors donnée par $\sigma_{[0..1[}\sigma^*_{[1..5[}$. La figure III-1-10 représente l'ordonnancement de ce système par l'algorithme *earliest deadline*.

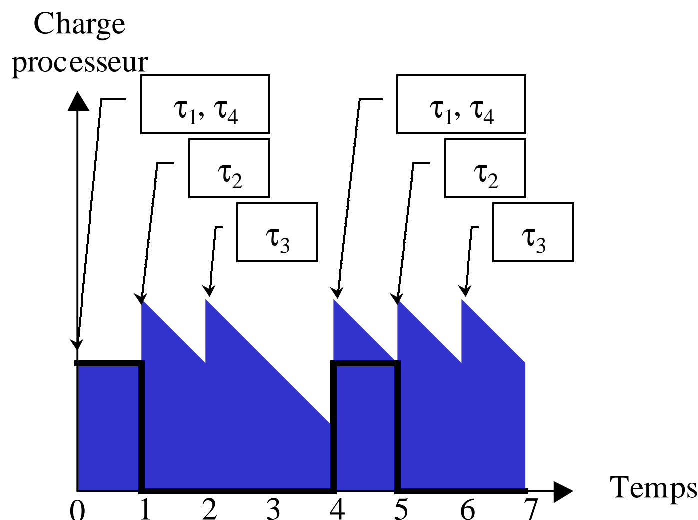


figure III-1-9 : diagramme de charge du système donné dans l'exemple III-1-3, le trait gras représente C_{att}

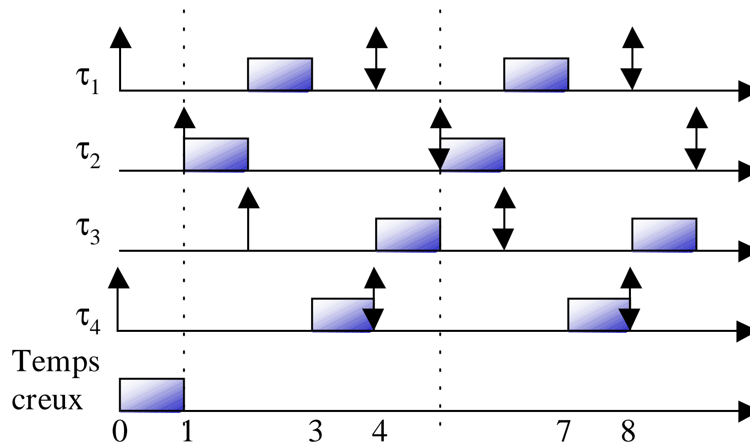


figure III-1-10 : séquence produite par earliest deadline pour l'exemple III-1-3

Les temps creux apparaissent donc lorsque la requête processeur croise la charge en attente. Les temps creux acycliques dus à l'absence de charge, comme nous en avons vu dans le cas des systèmes de tâches indépendantes, sont donc un cas particulier des temps creux dus aux chaînes d'attente avec $C_{att}=0$. Certains systèmes de tâches peuvent posséder ces deux types de temps creux, comme l'illustre l'exemple suivant.

---

exemple III-1-4 : Soit $S=\{\tau_1<0,1,4,4>, \tau_2<1,1,4,4>, \tau_3<5,2,4,4>\}$ un système de tâches du contexte $|1|r_i,C_i,D_i,P_i,prec|$, avec :

```
Tâche τ1 est
-- Attend un message
-- de τ2, puis dure une
-- unité de temps
r1 = 0 ;
P1 = 4 ;
D1 = 4 ;
Début
  Retirer_BAL(MB2,1) ;
  b1,1=1 ;
Fin ;
```

```
Tâche τ2 est
-- Dure une unité de
-- temps puis émet un
-- message à τ1
r2 = 1 ;
P2 = 4 ;
D2 = 4 ;
Début
  b2,1=1 ;
  Déposer_BAL(MB2,1) ;
Fin ;
```

```
Tâche τ3 est
-- Dure deux unités de temps
r3 = 5 ;
P3 = 4 ;
D3 = 4 ;
Début
  b3,1=2 ;
Fin ;
```

Le calcul des chaînes d'attente donne $C_{att}(0)= C_{att}(4)= C_{att}(8)=1$, et C_{att} est nul pour les autres dates comprises entre 0 et $r+P=9$. Le diagramme de charge donné sur la figure III-1-11 montre qu'il y a des temps creux acycliques aux instants 0, 3 et 4. Ces temps creux sont soit dus à des chaînes d'attente, soit dus à une absence de charge, ce qui est un cas particulier du cas précédent. Nous en déduisons $t_c=4$. Toute séquence conservative σ valide sur [0..9[est valide : la séquence infinie est alors donnée par $\sigma_{[0..5[}\sigma^*_{[5..9[}$.

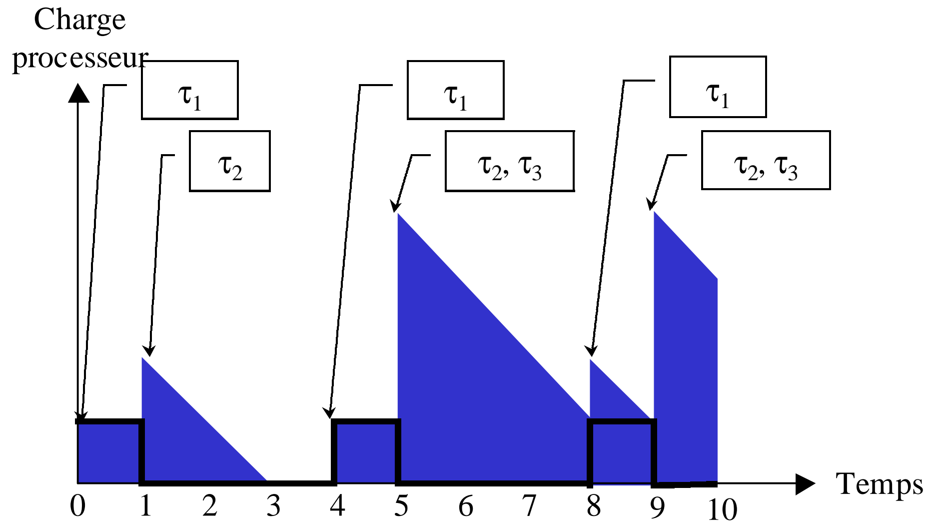


figure III-1-11 : diagramme de charge du système donné dans l'exemple III-1-4, le trait gras représente C_{att}

Remarque : Le calcul de t_c est donc, comme dans le cas des tâches indépendantes, totalement indépendant de l'algorithme d'ordonnancement choisi. Il est assujetti aux chaînes d'attente, dont le calcul n'est pas trivial. Cependant, il n'est pas nécessaire de calculer les chaînes d'attente, qui étaient instrumentales pour la preuve de cyclicité. En effet, les temps creux acycliques sont donnés par n'importe quel algorithme conservatif (par exemple *Rate Monotonic*) ne prenant en compte que les contraintes de précédence, puisque les ressources critiques n'entrent pas en jeu sous l'hypothèse qu'une tâche ne puisse pas attendre de message à l'intérieur d'une section critique.

III-2 Conclusion

Nous avons montré que les ordonnancements conservatifs (on dit aussi au plus tôt) de systèmes de tâches de charge *100%* du contexte $|1|ri,C_i,D_i,P_i, prec,resRW|$ étaient cycliques de période P après le dernier temps creux acyclique qui se trouvait dans tous les cas avant la date $r+P$. L'intérêt de notre preuve est qu'elle ne se base que sur la charge d'un système de tâches, et permet ainsi d'éviter toute distinction inhérente à un algorithme d'ordonnancement ou bien à une politique de gestion de ressources.

Notre théorème est d'une part un outil permettant de connaître la durée nécessaire à l'étude d'une séquence d'ordonnancement, qu'il s'agisse de la calculer hors-ligne, ou bien de réaliser une étude de performance hors-ligne d'une séquence qui sera produite en-ligne.

C'est une généralisation du théorème de Leung et Merrill, qui donne une durée de simulation nécessaire de $r+2P$ pour l'étude des systèmes de tâches indépendantes ordonnancées par *ED* dans le contexte $|1|r_i,C_i,D_i,P_i|$. En effet, notre résultat ne dépend pas de l'algorithme choisi. De plus il améliore le résultat de [LM 80] puisque $r+2P$ est maintenant une borne supérieure de la durée de simulation nécessaire.

On peut remarquer que ce résultat unifie les durées de simulation des systèmes de tâches simultanées et des systèmes dans lesquels des tâches sont différées. En effet, il n'y a aucun temps creux acyclique lorsque les tâches sont simultanées, donc t_c=-1, d'où une durée de simulation de P donnée par le théorème.

Cependant, la preuve est donnée sur des systèmes de charge *100%*, alors que ce n'est pas le cas la plupart du temps, de plus, les temps creux y sont placés au plus tard en utilisant une politique conservative (i.e. les tâches sont placées au plus tôt). Nous montrons maintenant pourquoi il reste valide dans le cas général où la charge d'un système est inférieure à *100%.*

III-2.1 Cas des algorithmes conservatifs

Lorsqu'un algorithme conservatif (i.e. au plus tôt) est utilisé, les temps creux sont tous placés au plus tard, qu'ils soient cycliques ou acycliques. Soit $S=\{\tau_i<r_i,C_i,D_i,P_i>\}_{i=1..n}$ un système de tâches temps réel de charge inférieure à 1 ordonnancé par un algorithme conservatif. D'après la définition de la charge d'un système, la séquence construite possède des temps creux cycliques et des temps creux acycliques. Affranchissons nous des temps creux cycliques en adjoignant au système une tâche englobant les temps creux cyclique, que nous nommons **tâche oisive**. Cette tâche, dénotée τ_0, a une période et un délai critique de P, et une charge de $P(1-U_S)$ puisque l'on sait, d'après la définition de la charge, qu'il y a exactement $P(1-U_S)$ temps creux toutes les P unités de temps en régime permanent, c'est à dire dans la partie cyclique de l'ordonnancement. Pour l'instant, nous ne nous intéressons pas à sa date de réveil, que nous laissons délibérément de côté. La séquence d'ordonnancement produite pour un système S de charge inférieure à 1 par un algorithme conservatif est identique à la séquence produite par le même algorithme d'un système $S'=S \cup\{\tau_0<r_0,P(1-U_S),P,P>\}$ pour lequel la priorité de τ_0 est minimale (la tâche oisive remplace alors exactement les temps creux cycliques). Nous avons donc transformé de façon artificielle le système S en un système S' de charge 100%. Le théorème portant sur la cyclicité s'applique donc à ce cas.

Le dernier point à éclaircir est le choix de la date de réveil de la tâche oisive, qui conditionnera la différenciation entre temps creux cycliques et acycliques, c'est à dire le choix de la date t_c. Il est important de remarquer que rien n'oblige la tâche oisive à avoir une date de réveil nulle. Conceptuellement, les temps creux, qu'ils soient cycliques ou acycliques, sont

identiques, c'est à dire qu'ils représentent l'oisiveté du processeur. On peut donc choisir arbitrairement une date de réveil pour la tâche oisive. L'exemple III-2-1 illustre notre propos.

exemple III-2-1 : Soit $S=\{\tau_1<0,1,4,4>, \tau_2<4,3,6,6>\}$ un système de tâches du contexte $|1|r_i,C_i,D_i,P_i|$. Sa charge U_S est de *0.75* donc il y a $0.25\times12=3$ temps creux toutes les $P=12$ unités de temps lorsque le système est en régime permanent.

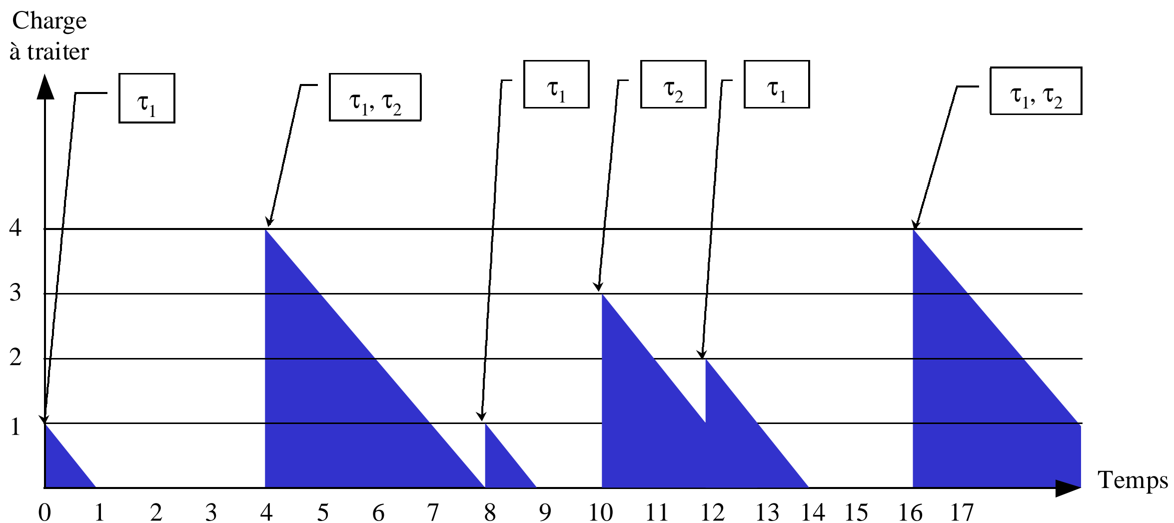


figure III-2-1 : diagramme de charge d'un système

La figure III-2-1 donne le diagramme de charges de *S*, sur lequel on distingue *6* temps creux entre *0* et $r+P$. Si l'on fait débuter la tâche oisive à l'instant *0*, comme on pourrait être tenté de le faire a priori, le dernier temps creux acyclique apparaît à l'instant *9*, la longueur de la séquence à construire est donc de $9+1+12=22$ unités de temps, ce qui améliore peu la borne de Leung et Merrill (qui rappelons-le tout de même ne pourrait s'appliquer que si les tâches étaient indépendantes et ordonnancées à l'aide d'une politique *earliest deadline*) puisque $r+2P=26$. Cependant, rien ne contraint à choisir $r_0=0$. Choisissons donc, en vue de diminuer la longueur de la séquence requise, $r_0=2$. Dans ce cas, le premier temps creux est acyclique, et les temps creux suivants sont englobés par la tâche oisive. Nous obtenons t_c=1, et une longueur de séquence donnée par $1+1+12=14$, contre *26* précédemment.

Il convient donc de faire démarrer la tâche oisive après l'instant t_c puisque c'est après cet instant que le système entre en régime permanent. Il faut de plus, que le temps creux suivant t_c soit considéré comme un temps creux cyclique, afin de ne pas augmenter artificiellement la partie transitoire de la séquence. Soit *t'* la date d'occurrence du temps creux suivant t_c, alors la meilleure date de réveil pour τ_0 est comprise dans l'intervalle $]t_c..t]$. L'instant t_c+1 satisfait toujours cette condition, c'est donc cette date que nous choisirons comme date de réveil pour τ_0.

Il nous reste maintenant à proposer un algorithme calculant t_c intégrant le fait que $r_0=t_c+1$. Considérons les temps creux présents dans $[0..r+P[$. Si il y en a C_0, alors ils sont considérés cycliques et t_c=-1. Si il y en a plus que C_0, alors il est possible que certains d'entre eux soient acycliques : considérons le premier temps creux, positionné à l'instant t_1 et voyons combien de temps creux sont présents dans l'intervalle $[t_1..t_1+P[$. Si il y en a plus que C_0, alors il y a au moins un temps creux acyclique dans cet intervalle, nous décidons donc que le premier est acyclique. Le même processus est alors répété avec les temps creux suivants, jusqu'à ce qu'il ne reste plus que C_0 temps creux, lesquels peuvent être attribués à la tâche oisive. Donc les

temps creux situés avant t_c+1 sont acycliques, et les temps creux situés après sont cycliques. Nous choisissons donc arbitrairement, dans le but de faire apparaître le cycle plus rapidement, d'adjoindre au système la tâche oisive $\tau_0<t_c+1,P(1-U_S),P,P>$. L'algorithme suivant illustre le calcul de la position des temps creux et le choix de t_c :

```
Fonction DernierTempsCreuxAcyclique renvoie date est
TempsCreux: liste de dates ← ∅
Crest: charge ← Creq(0)
tc: date
Début
      Pour t allant de 0 à r+P-1 faire
      -- Calcul de la position des temps creux
            Si Crest=Catt(t) alors ajouter t à la fin de TempsCreux
            Sinon Crest ← Crest-1
            FinSi
            Crest ← Crest+Creq(t+1)
      Fait
      tc ← -1
      Tant que longueur(TempsCreux)>P(1-US) faire
            t1 ← tête(TempsCreux)
            t2 ← (P(1-US)+1)ème élément de TempsCreux
            Si t2-t1<P alors
            -- Il y a plus de P(1-US) temps creux dans [t1..t1+P[
            -- t1 est donc un temps creux acyclique
                  tc ← t1
                  SupprimerTête(TempsCreux)
            FinSi
      Fait
      Renvoyer tc
Fin
```

Cet algorithme n'est pas polynomial puisqu'il nécessite la construction du diagramme de charge sur $r+P$, avec P non polynomial. Il serait intéressant de trouver une méthode analytique polynomiale permettant de calculer t_c, mais nous n'avons pas trouvé pour le moment une telle méthode.

---

exemple III-2-2 : Soit $S=\{\tau_1<0,1,4,4>, \tau_2<1,1,4,4>, \tau_3<0,1,4,4>\}$ un système de tâches du contexte $|1|r_i,C_i,D_i,P_i,prec|$, le pseudo-code des tâches est le suivant :

```
Tâche τ1 est
-- Attend un message
-- de τ2, dure une unité de
-- temps puis émet un
-- message à τ3
r1 = 0 ;
P1 = 4 ;
D1 = 4 ;
Début
  Retirer_BAL(MB2,1)
  b1,1=1 ;
  Déposer_BAL(MB1,3) ;
Fin ;
```

```
Tâche τ2 est
-- Dure une unité de
-- temps puis émet un
-- message à τ1
r2 = 1 ;
P2 = 4 ;
D2 = 4 ;
Début
  b2,1=1 ;
  Déposer_BAL(MB2,1)
Fin
```

```
Tâche τ3 est
-- Attend un message de τ1 puis
```

```
-- dure une unité de temps
r3 = 0 ;
P3 = 4 ;
D3 = 4 ;
Début
  Retirer_BAL(MB1,3)
  b3,1=1 ;
Fin ;
```

La charge du système est U_S=*0.75* donc il y a *0.25*×*4*=1 temps creux toutes les *P*=*4* unités de temps lorsque le système est en régime permanent. La charge en attente est définie par $C_{att}(0)=C_{att}(5)=2$, et elle est nulle ailleurs entre *0* et *5*. Le diagramme de charge de ce système est donné sur la figure III-2-2 : il contient deux temps creux entre 0 et 5. Cependant, l'écart qu'il y a entre les deux est supérieur à *P*, donc il n'y a pas de temps creux acyclique et t_c=-1.

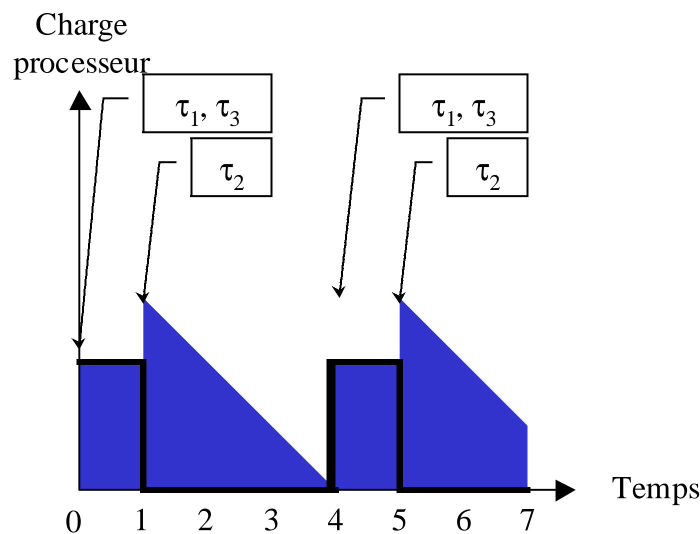


figure III-2-2 : diagramme de charge de l'exemple III-2-2, la charge en attente est représentée en traits gras

III-2.2 Cas des algorithmes non conservatifs

Nous avons vu en section I-2.2.3.d que les algorithmes conservatifs n'étaient pas dominants (i.e. ne fournissaient pas forcément une séquence valide lorsqu'il en existait une) dès lors que des ressources ou des parties non préemptibles existaient. Il en résulte que certaines techniques d'ordonnancement (et notamment celle que nous proposons) doivent être à même d'ordonnancer les systèmes de façon non conservative (i.e. les temps creux ne doivent pas être forcément placés au plus tard). La liberté de placer les temps creux cycliques de façon non conservatives peut être artificiellement prise en compte par la tâche oisive que nous avons vue dans la section précédente. La différence par rapport au cas des algorithmes conservatifs est que cette tâche n'est pas forcément de priorité minimale dans le système. L'adjonction à un système d'une tâche oisive (qui n'est pas forcément dans ce cas de priorité minimale) permet donc aux temps creux cycliques d'être placés de façon non conservative. De plus, elle permet comme nous l'avons vu d'augmenter artificiellement la charge du système de tâches à 100%. Enfin, le choix de faire démarrer la tâche oisive de à la date t_c+1 implique que la durée de simulation choisie est minimale.

Un dernier point est à éclaircir : la tâche oisive permet de placer les temps creux cycliques de façon non conservative sans rien changer à la preuve de cyclicité. Cependant, nous voyons sur la figure III-2-3 qu'il est aussi primordial de pouvoir placer les temps creux acycliques de façon non conservative. Il nous faut donc montrer que le fait de placer les temps creux acycliques qui apparaissent lors de la montée en charge de façon non conservative ne modifie en rien la cyclicité des séquences d'ordonnancement.

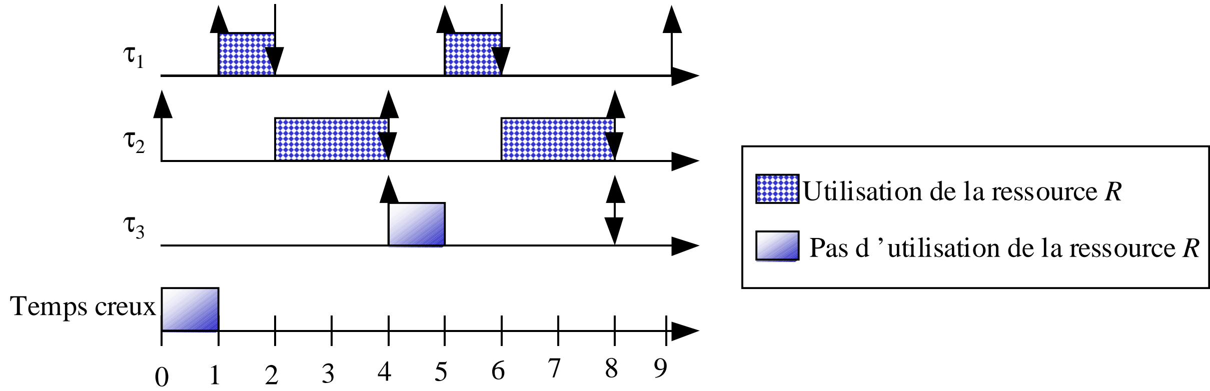


figure III-2-3 : un ordonnancement valide d'un système de tâches $S=\{\tau_1<1,1,1,4>, \tau_2<0,2,4,4>, \tau_3<4,1,4,4>\}$ avec τ_1 et τ_2 partageant une ressource critique R. Il est nécessaire de placer le temps creux acyclique de façon non conservative pour que ce système soit ordonnançable.

Le diagramme de charge de ce système est donné sur la figure III-2-4, sur lequel on trouve un temps creux acyclique à la date *3*, d'où $t_c=3$.

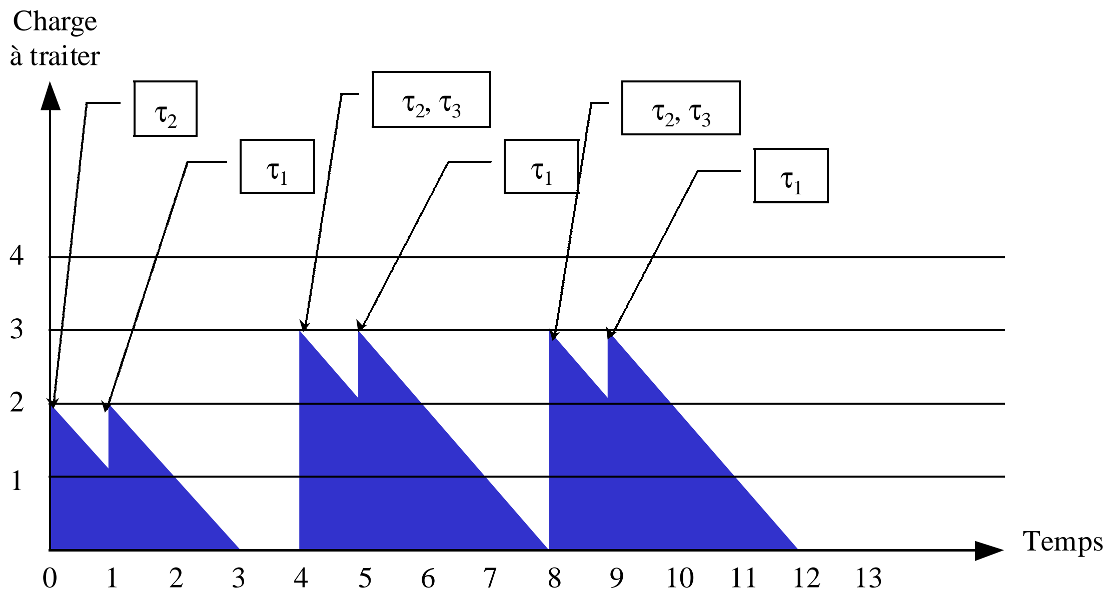


figure III-2-4 : diagramme de charge du système $S=\{\tau_1<1,1,1,4>, \tau_2<0,2,4,4>, \tau_3<4,1,4,4>\}$.

Or, nous voyons que la charge du système évolue, dans la séquence produite sur la figure III-2-3, de la façon illustrée sur la figure III-2-5.

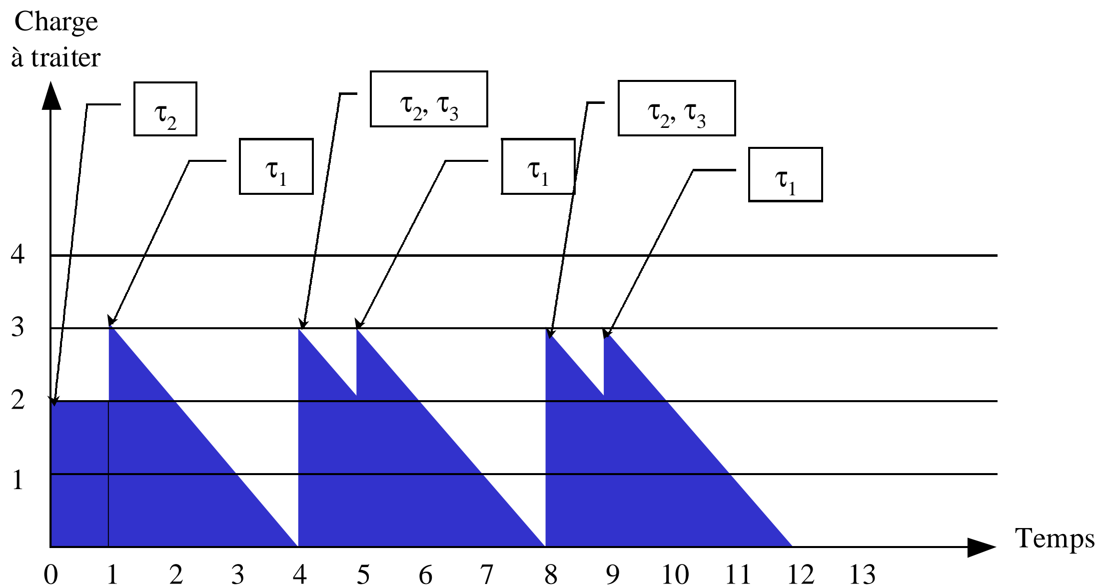


figure III-2-5 : diagramme de charge correspondant à la séquence produite sur la figure III-2-3.

Sur cette figure, le temps creux acyclique est placé à l'instant *0*, ce qui se traduit par un seuil de charge. Voyons en détail ce que dit la Proposition III-1-1 : l'état à l'instant suivant le dernier temps creux acyclique (quand ceux-ci sont placés de façon conservative), c'est à dire ici à l'instant *4* est le même qu'une méta-période suivante : il correspond à la réactivation des tâches (plus éventuellement la charge en attente que l'on n'a pas pu traiter) à cet instant. Le fait de permettre aux temps creux acycliques d'intervenir plus tôt dans les ordonnancements ne change aucunement ce fait. Le théorème reste donc valide dans ce cas, cependant, le cycle intervient plus tôt dans ce cas (sur l'exemple de la figure III-2-5, le cycle commence à l'instant 1, alors que le théorème donne la date *4*). En effet, le choix de positionner les temps creux acycliques de façon non conservative place ceux-ci plus tôt que t_c. La borne t_c+P+1, basée sur le fait que la charge à l'instant t_c+1 est exactement donnée par les mêmes activations qu'à l'instant t_c+P+1 tient toujours, mais n'est plus qu'une borne supérieure.

Le résultat de cyclicité s'applique donc aux méthodes non conservatives qui permettent au temps creux d'apparaître autrement qu'au plus tard. Il permet d'appliquer les techniques hors-ligne d'ordonnancement de tâches valides dans le contexte non périodique $|1|\ ri,C_i,D_i, prec,resRW/$ au contexte périodique $|1|ri,C_i,D_i,P_i, prec,resRW|$: puisque le domaine d'étude se limite à $[0..t_c+P+1[$, il suffit de considérer les $\left\lceil \frac{t_c+P+1-r_i}{P_i} \right\rceil$ premières occurrences de chaque tâche τ_i sauf celle qui a lieu à la date t_c+P+1 si il y en a une, puisqu'elle correspond dans ce cas au début du deuxième cycle de l'ordonnancement.

IV Etude de Systèmes Temps Réel à l'Aide de Réseaux de Petri: Cas Monoprocesseur

Notre objectif est d'exhiber des séquences d'ordonnancement valides, quel que soit le système de tâches modélisé. En effet, les tâches considérées peuvent communiquer, partager des ressources de façon exclusive ou en lecture-écriture, et posséder des parties non préemptibles. Les séquences exhibées pourront avoir certaines propriétés qualitatives (optimalité suivant certains critères), et quantitatives (respect de contraintes absolues basées principalement sur le temps de réponse). Notre méthodologie repose sur une technique d'énumération basée sur un réseau de Petri : le système de tâches à étudier est modélisé finement par un réseau de Petri avec la règle de tir maximal, prenant en compte les interactions complexes entre les tâches. Cette « implémentation », que nous décrivons ici dans le cas monoprocesseur, s'avère très flexible.

IV-1 Modélisation

IV-1.1 Hypothèses générales

Nous présentons dans cette section la modélisation de tâches périodiques sur lequel est basé le modèle initialement proposé dans [CGC 96]. Nous avons vu que, dans le cas monoprocesseur, les techniques en-ligne d'ordonnancement étaient limitées notamment lorsque des ressources critiques étaient en jeu, et aussi parce que le système de tâches à étudier devait être mis sous forme normale. La tâche dont une partie du pseudo code est donnée dans l'exemple IV-1-1 ne peut pas être mise sous forme normale, dans le cas contraire, la ressource serait prise par une tâche et rendue par une autre.

exemple IV-1-1 : pseudo code d'une tâche ne pouvant pas être mise sous forme normale.

```
Prendre_Sémaphore(ressource,1)
Déposer_BAL(bal,msg)
Rendre_Sémaphore(ressource,1)
```

L'utilisation d'un RdP permet de modéliser finement les tâches sans avoir à les éclater. Chaque tâche du système de tâches à modéliser est représentée par une branche périodique, alors que le corps des tâches est modélisé directement à partir du pseudo code de celle-ci. Le pseudo code utilisé se veut le plus proche possible des instructions proposées dans les noyaux temps réel. La notion d'événement est délibérément mise de côté à cause du problème posé dans l'exemple I-1-1 (page 13). Par contre, la notion de boîte aux lettres, facilement modélisable par RdP, est retenue afin de gérer les communications entre tâches. Ce modèle de communication relativement large permet de prendre en compte les communications entre tâches de périodes différentes. Rappelons que nous ne considérons que les systèmes de tâches dont les fréquences d'émission et de réception sont corrélées, c'est à dire que l'équation suivante doit être vérifiée par les systèmes de tâches modélisés :

$$\frac{\textit{nombre de messages émis}}{\textit{période tâche émettrice}} = \frac{\textit{nombre de messages reçus}}{\textit{période tâche réceptrice}}$$

Cette hypothèse est moins contraignante que celle qui est classiquement utilisée en ordonnancement en-ligne, dans laquelle les périodes des tâches communicantes sont identiques.

Cette contrainte peut être partiellement relâchée en considérant que plusieurs tâches émettrices et réceptrices peuvent partager la même boîte aux lettres. Cela permet de modéliser entre autres la multidiffusion (dans le cas où plusieurs réceptrices partagent la même boîte aux

lettres) pourvu que la tâche émettrice sache combien de réceptrices doivent recevoir son message. Dans ce cas, la contrainte devient :

$$\sum_{\tau_i \text{ émettrice}} \frac{\text{nombre de messages émis par } \tau_i}{P_i} = \sum_{\tau_i \text{ réceptrice}} \frac{\text{nombre de messages lus par } \tau_i}{P_i}$$

De plus nous avons vu que le rendez-vous peut être modélisé à l'aide de deux boîtes aux lettres. La notion de sémaphore est généralisée dans le cadre de notre étude aux accès aux ressources en lecture ou en écriture [ZRS 87b].

Puisque nous voulons produire des ordonnancements valides de systèmes de tâches temps réel, nous ne nous intéressons qu'à l'aspect temporel des tâches. Ainsi, le code composant les tâches est abstrait aux durées des instructions. Cependant, les interactions entre les tâches, qui induisent des relations de précédence ou d'exclusion entre certaines parties des tâches, doivent être prises en compte. Ces interactions sont les primitives de communication par boîtes aux lettres ou de synchronisation par sémaphores, appelées primitives temps réel ou **PTR.** Le pseudo code d'une tâche peut alors être vu comme une succession de blocs dits indépendants (correspondant aux instructions classiques) car sans PTR, entrecoupés par des PTR. Les PTR étant considérées de durée nulle, leur durée véritable est intégrée à la durée du bloc indépendant précédent.

Le temps est discrétisé de sorte qu'un quantum de temps corresponde à un nombre entier de points de préemptions du système d'exploitation sous-jacent. L'unité de temps prise comme base est donc la durée d'un quantum de préemption du noyau sous-jacent.

La durée d'un bloc est alors calculée en nombre de quanta de temps, en arrondissant au quantum de temps supérieur, et en ajoutant la durée des surcoûts pouvant être engendrés par la durée des changements de contexte en cas de préemption. Ensuite, chaque boucle contenant des PTR est déroulée (dans un contexte temps réel, le nombre d'itérations d'une boucle est borné à l'avance), alors que les boucles indépendantes sont abstraites à leur durée maximale et les branches conditionnelles sont réunies. Dans le cas où les branches de la conditionnelle sont composées d'instructions indépendantes, cela paraît judicieux, mais des problèmes de cohérence peuvent se poser lorsque les branches comportent des PTR : dans ce cas, on fait généralement l'union structurelle des deux branches (voir chapitre 7 de [Bab 96]).

Par rapport aux instructions classiques d'un noyau temps réel, notre modèle permet de modéliser toutes les instructions données dans le tableau IV-1-1. Il faut noter que puisque nous nous plaçons dans le cadre d'une analyse hors-ligne, toutes les tâches, ressources et boîtes aux lettres sont connues à l'avance, et les primitives de création de telles entités ne sont pas autorisées dans le corps des tâches. En effet, tout est créé dès le début de l'application.

Types de primitives modélisées	Contraintes imposées au concepteur
Communication par boîtes aux lettres (envoi et réception de messages)	Mêmes fréquences d'émission et de réception de messages vers une boîte aux lettres
Prises et libérations de ressources, les prises peuvent être effectuées en lecture seule	Une tâche ne doit pas attendre de message à l'intérieur d'une section critique

tableau IV-1-1 : primitives temps réel prises en compte dans notre modèle

Typiquement, le pseudo code d'une tâche est de la forme de celle donnée sur l'exemple IV-1-2.

exemple IV-1-2 : pseudo code d'une tâche τ_i

```
bi,1=3
-- Cette durée représente la durée du 1er bloc de τi,
-- elle inclut la durée de la PTR suivante
Retirer_BAL(bal1)
-- Le message n'est pas pris en compte, et la durée induite
-- par son envoi est incluse dans la durée précédente
bi,2=2
Lock(res1,1,lecture_seule)
-- Prise d'une ressource en lecture seule
bi,3=2
Déposer_BAL(bal2)
Unlock(res1,1)
-- Libération de la ressource
```

Dans la suite de cette partie, le système de tâches S à modéliser est défini par $S=\{\tau_{i,i=0..n}<r_i,C_i,D_i,P_i>\}$ avec τ_0 tâche oisive telle qu'elle a été définie précédemment : $\tau_0<t_c+1,P(1-U_S),P,P>$. De plus, afin que les temps creux acycliques puissent être placés de façon non conservative, une place contenant un nombre de jetons égal au nombre de temps creux acycliques présents dans une séquence, couplée à une transition permettant de modéliser les temps creux acycliques sera adjointe au système.

IV-1.2 Principe général

La modélisation se base sur la construction automatique d'un réseau de Petri à partir du pseudo-code d'un système de tâches temps réel du contexte $|1|r_i,C_i,D_i,P_i,prmptnoprmpt,prec,resRW|$. Ce réseau de Petri prend en compte le temps de par sa règle de tir maximal, il a donc la puissance d'expression nécessaire à la modélisation fine d'un programme informatique.

Il est composé d'une partie horlogerie, dans laquelle une transition source, nommée *horloge globale*, marque le temps logique en produisant des jetons dans les horloges locales des tâches. Les horloges locales permettent la réactivation périodique des tâches du système et la vérification du respect des contraintes temporelles qui leur sont imposées.

Enfin, la partie système de tâches du modèle représente finement le pseudo-code décrivant les tâches, et notamment les contraintes structurelles imposées par les PTR : contraintes de précédence et d'exclusion. Le modèle est conçu de sorte qu'à chaque instant, une transition du système de tâches puisse être franchie, modélisant ainsi le travail effectué par le processeur.

Une vue abstraite du modèle est représentée sur la figure IV-1-1.

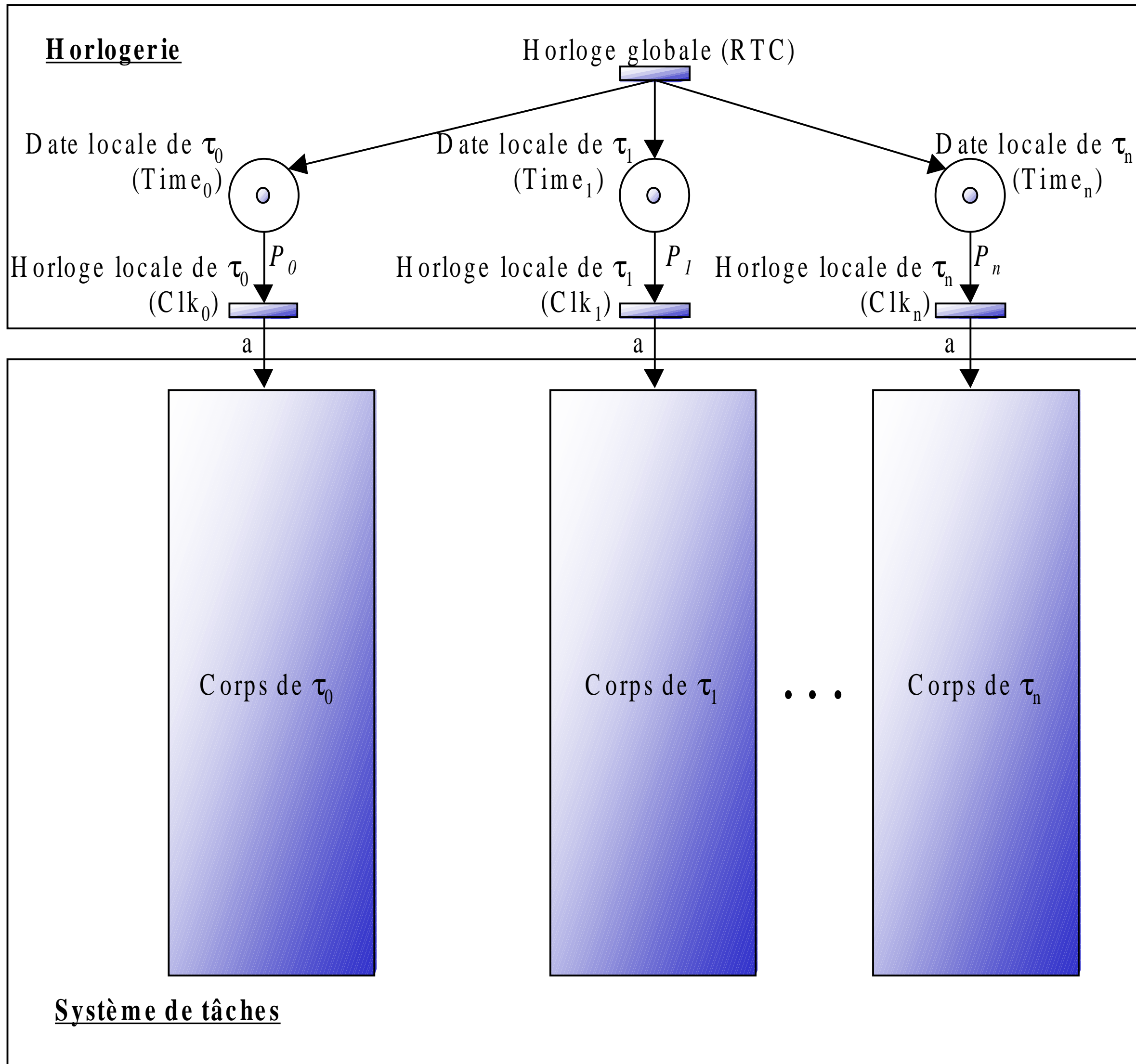


figure IV-1-1 : schéma d'ensemble de la modélisation par réseau de Petri

Le détail de notre modélisation est donné dans les paragraphes suivants.

IV-1.3 Système d'horlogerie

Nous considérons un ensemble de tâches périodiques, les autres tâches à contraintes strictes étant prises en compte à l'aide de serveurs périodiques. Il nous faut donc représenter la périodicité d'une action à l'aide d'un réseau de Petri avec la règle de tir maximal. La figure IV-1-2 représente un réseau de Petri dans lequel la transition *Action* est tirée toutes les 8 unités de temps.

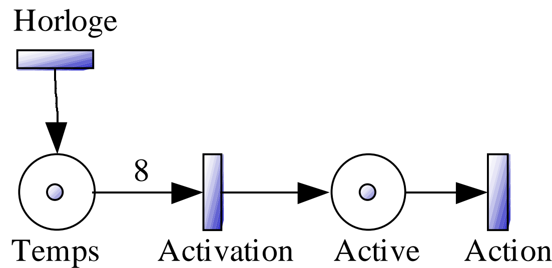


figure IV-1-2 : modèle pour une action périodique de période 8

La transition *Horloge* est une transition source, qui est toujours franchissable : elle marque donc le temps logique. En effet, à partir de $M_0=\{1,1\}$, les transitions *Horloge* et *Action* sont franchissables simultanément. Elle sont donc franchies simultanément conformément à la règle de tir maximal, et leur tir mène à $M_1=\{2,0\}$. Seule la transition *Horloge* est alors franchissable, et son franchissement mène à $M_2=\{3,0\}$, etc… jusqu'au marquage $M_7=\{8,0\}$ après 7 tirs successifs de la transition *Horloge*. A partir de M_7, 2 transitions sont franchissables simultanément : *Activation* et *Horloge*. Elles sont donc franchies simultanément et mènent à $M_0=\{1,1\}$. Exactement 8 tirs de la transition *Horloge* sont nécessaires pour retrouver M_0 qui est un marquage d'accueil du réseau. La transition *Action* est donc tirée à l'unité de temps 1, 9, 17, etc…

C'est ce principe qui est utilisé pour modéliser la périodicité des tâches dans la partie horlogerie du modèle, qui est représentée en non grisé sur la figure IV-1-3.

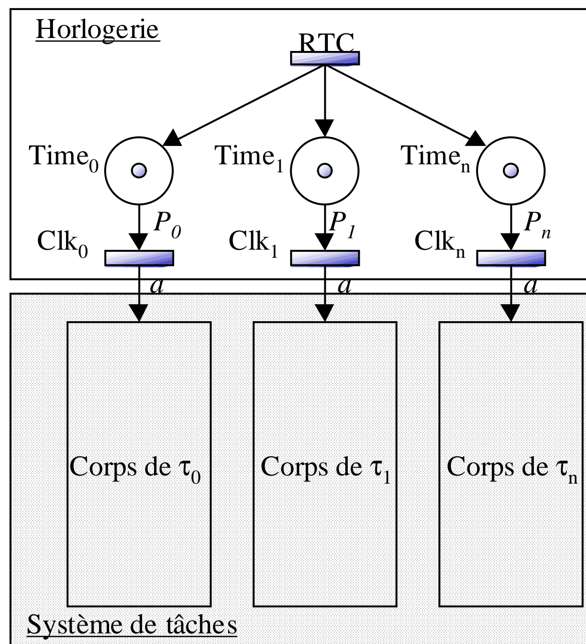


figure IV-1-3 : partie horlogerie du modèle

La partie horlogerie se base sur la transition *RTC* pour *Real-Time Clock* qui joue le rôle de la transition *Horloge* de la figure IV-1-2, c'est à dire de produire un jeton à chaque unité de temps dans chacune des horloges locales des tâches, en l'occurrence dans chacune des places $Time_i$. Etant donné la règle de tir du réseau, les horloges locales obligent les transitions d'activation Clk_i à tirer toutes les périodes P_i des tâches. Cela implique qu'un jeton de couleur a (pour activation) est produit toutes les P_i unités de temps dans une place du corps de τ_i.

Cette place se nomme $activ_i$, et on dit qu'une tâche τ_i est activée lorsque la place $activ_i$ reçoit un jeton de couleur a. Nous expliquerons plus loin en détail l'utilité de ce jeton, et le fait qu'il soit coloré, il faut seulement ici retenir que le système d'horlogerie produit périodiquement un jeton de couleur a dans $activ_i$.

IV-1.4 Systèmes de tâches

Dans cette partie, la modélisation du corps des tâches est détaillée. Cela correspond donc à la partie inférieure, nommée système de tâches, du modèle représenté sur la figure IV-1-4.

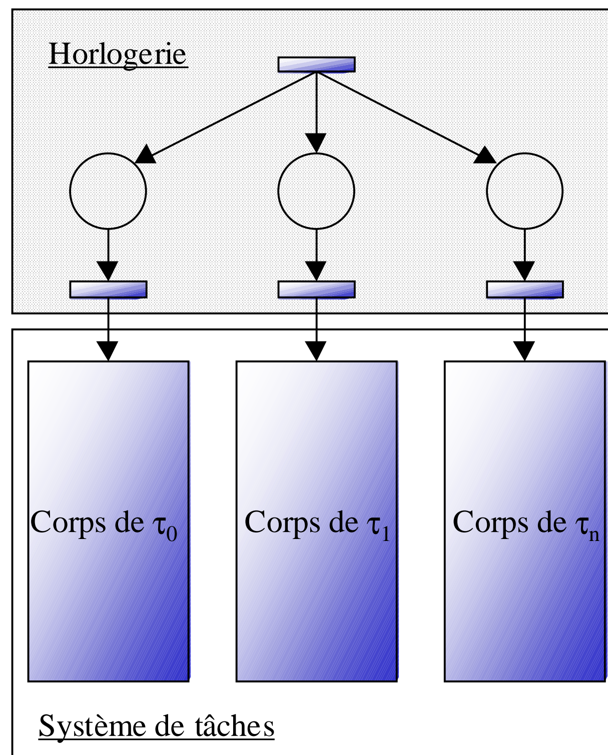


figure IV-1-4 : la partie structurelle des tâches est la partie inférieure

IV-1.4.1 Modélisation des blocs indépendants

Puisqu'avec la règle de tir maximal, une unité de temps est nécessaire au franchissement d'une transition, on peut modéliser un bloc de durée n par une chaîne de n transitions. Cependant, le modèle que nous utilisons permet d'utiliser au plus 3 transitions par bloc, le principe est explicité sur la figure IV-1-5.

Rappelons qu'une tâche est composée de blocs indépendants séparés par des PTR, avec $b_{i,j}$ la durée $j^{ème}$ bloc indépendant de la tâche τ_i. Le processeur doit donc consacrer $b_{i,j}$ quanta de temps au bloc pour le traiter.

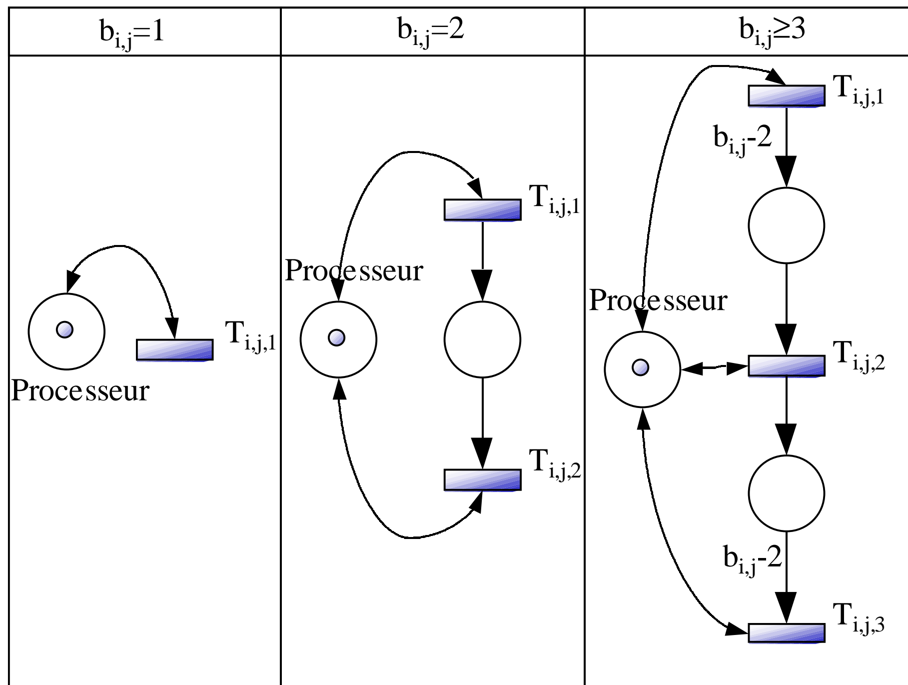


figure IV-1-5 : modélisation du $j^{ème}$ bloc indépendant de τ_i, une double flèche sur un arc représente l'union de deux arcs : un dans chaque sens

Une transition est utilisée pour modéliser un bloc d'une unité de temps : il faut donc au RdP une unité de temps pendant laquelle le processeur est utilisé pour tirer la transition. Les blocs de durée 2 sont modélisés par deux transitions mises en séquence, il faut donc au RdP 2 unités de temps pour les tirer. Les blocs de durée $b_{i,j} \geq 3$ unités de temps sont modélisés par trois transitions, et la transition $T_{i,j,3}$ ne peut tirer qu'une fois que $T_{i,j,1}$ a tiré une fois et que $T_{i,j,2}$ a tiré $b_{i,j}-2$ fois, la séquence de transition nécessite donc $b_{i,j}$ unités de temps. Ces trois transitions sont nécessaires à la prise en compte des préemptions pouvant intervenir à chaque quantum de temps.

Il est important de remarquer que dans le cas monoprocesseur, une ressource est partagée par chaque action de durée unitaire des tâches : le processeur. Afin d'alléger les représentations graphiques des RdP à venir, on pourra omettre les arcs existants entre la place *Processeur* et chacune des transitions de la partie système de tâches.

La nécessité pour une transition du système de tâches d'utiliser la marque de la place processeur pour progresser illustre la nécessité pour une tâche d'utiliser le processeur pour progresser dans son exécution.

Il peut être intéressant de garantir la non préemptibilité d'une tâche ou d'une partie de tâche. Pour cela, il suffit de ne pas rendre le processeur entre la première transition de la partie non préemptible et la dernière : ainsi, aucune autre tâche ne pourra utiliser le processeur tant que la partie non préemptible n'est pas terminée.

Un exemple de tâches périodiques dont le pseudo code est donné dans l'exemple IV-1-3 est représenté sur la figure IV-1-6.

---

exemple IV-1-3 : pseudo code d'un système de 2 tâches périodiques indépendantes

```
Tâche τ1 est          Tâche τ2 est
r1 = 0 ;              r2 = 0 ;
D1 = 4 ;              D2 = 2 ;
P1 = 4 ;              P2 = 2 ;
Début                 Début
```

```
    b1,1=2 ;             b2,1=1 ;
  Fin ;                Fin ;
```

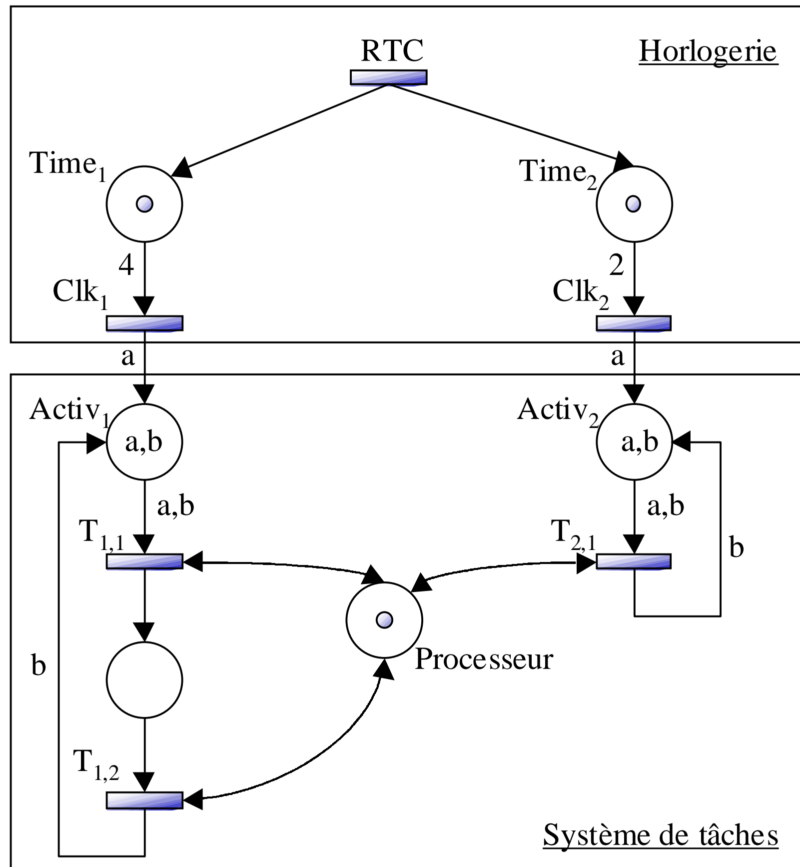


figure IV-1-6 : modélisation d'un système de tâches par réseau de Petri

IV-1.4.2 Modélisation des communications

Chaque boîte aux lettres est modélisée par une place. L'émission d'un message par une tâche vers cette boîte correspond au dépôt d'un jeton dans cette place, et la réception correspond à la consommation d'un jeton. La transition déposant un jeton est la transition appartenant au bloc précédent, en effet rappelons que les PTR sont supposées de durée nulle. La transition retirant un jeton est la transition correspondant à la première unité de temps du bloc suivant l'attente du message : cette transition ne peut être tirée que lorsque le message est disponible.

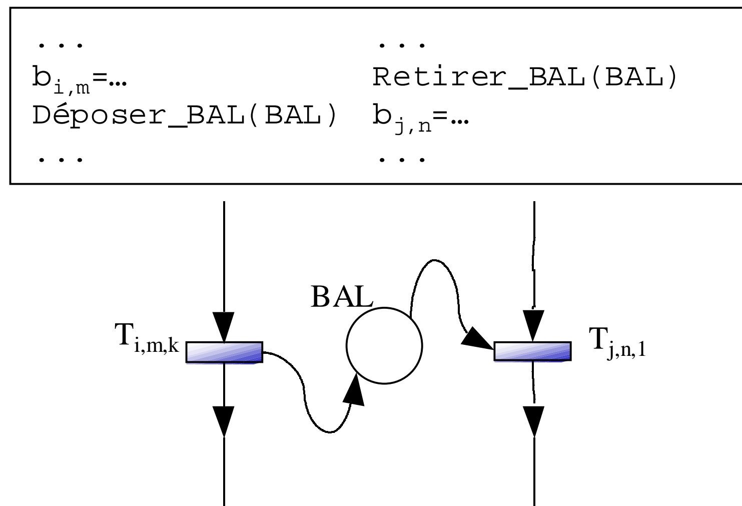


figure IV-1-7 : modélisation d'une communication par boîte aux lettres

La figure IV-1-7 représente une boîte aux lettres dans laquelle une tâche τ_i envoie un message après son bloc $b_{i,m}$. La tâche τ_i attend un message de cette boîte avant de commencer $b_{j,n}$.

On peut remarquer qu'une limitation inhérente au modèle est que le bloc précédent l'envoi d'un message ne peut pas être vide, mais cette limitation n'en est pas une en pratique puisque la PTR d'envoi d'un message n'est pas en général de durée nulle, comme cette durée est reportée dans le bloc précédent, ce bloc n'a pas une durée nulle. Cet argument peut être réfuté dans le cas où la granularité des tâches est grossière et où la durée des PTR est négligée, cependant, même dans ce cas, il est difficile d'imaginer une tâche qui envoie un message avant même d'avoir fait le moindre calcul.

De façon symétrique, on ne peut pas modéliser l'attente d'un message qui ne soit pas suivie d'un bloc de traitement. Cela n'est pas vu non plus comme une limitation car l'existence d'une tâche attendant un message sans faire aucun traitement postérieur n'est pas réaliste.

IV-1.4.3 Modélisation des ressources critiques

Le modèle RdP permet de modéliser simplement l'utilisation d'une ressource en exclusion mutuelle, comme c'est représenté sur la figure IV-1-8. Si la ressource R est disponible en m exemplaires, alors une tâche τ_i peut en utiliser $n_i \leq m$, et une tâche τ_j peut en utiliser $n_j \leq m$. Si $n_i + n_j \leq m$, il est possible que les sections critiques de τ_i et τ_j se préemptent mutuellement.

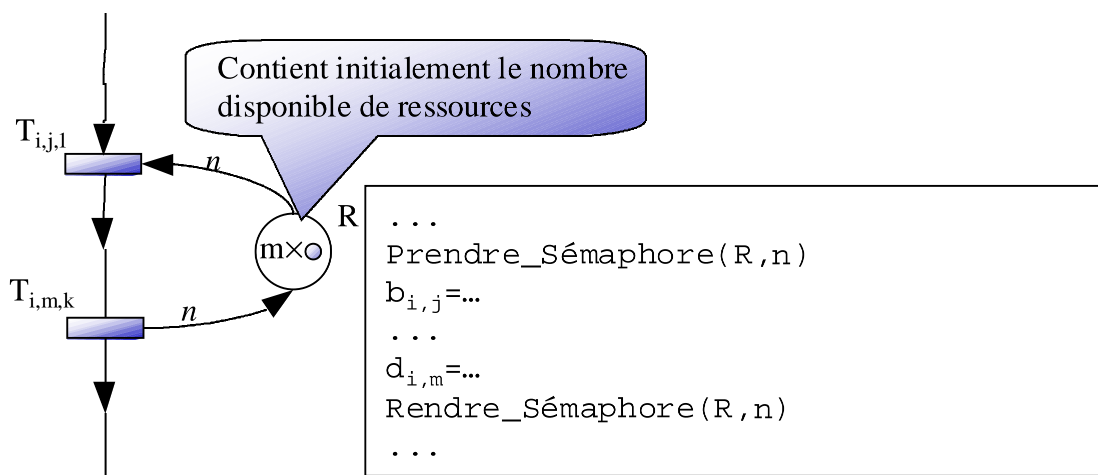


figure IV-1-8 : modélisation de l'utilisation d'une ressource critique

Dans le cas des ressources utilisables en lecture/écriture, on suppose généralement que les ressources ne sont disponibles qu'en un seul exemplaire, cependant plusieurs lecteurs peuvent y avoir accès simultanément, alors qu'un écrivain doit y accéder en excluant tout autre accès. Puisque le nombre maximal de lecteurs simultanés r_l de la ressource R est connu à l'avance, le marquage initial de la place ressource est r_l. Un lecteur ne prendra qu'un seul jeton pour accéder à R en lecture, alors qu'un écrivain prendra r_l jetons : on est assuré ainsi que les r_l lecteurs peuvent faire un accès simultané à R, alors que les écrivains ne peuvent accéder que seuls à R. Ce principe est explicité sur la figure IV-1-9.

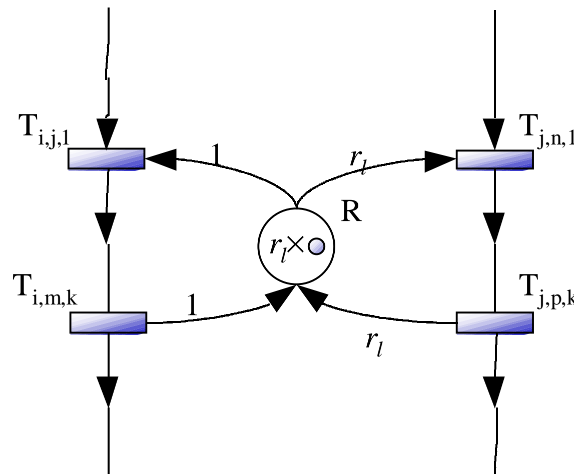


figure IV-1-9 : accès à une ressource par un lecteur (à gauche) et un écrivain (à droite)

IV-1.4.4 Schéma global d'une tâche

Une tâche composée de blocs indépendants séparés par des PTR est modélisée par une chaîne de transitions nécessitant toutes le processeur pour être franchies. Ces transitions peuvent être connectées à des places boîtes aux lettres ou ressources, modélisant ainsi les contraintes structurelles inhérentes aux PTR.

La place initiant la chaîne de transitions, nommée $activ_i$ peut contenir des jetons de couleur a signifiant que la tâche est active, on dit aussi que chaque jeton a modélise une requête de la tâche. De plus, cette place peut aussi contenir des jetons de couleur b, signifiant que la tâche n'est pas en cours d'exécution, on dit aussi que la tâche est au repos. On dira qu'une tâche τ_i est en cours d'exécution entre le tir de la première transition composant son corps et le tir de la dernière transition de son corps, correspondant à la terminaison de son exécution.

Par exemple, sur la figure IV-1-10, la tâche τ_1 est en cours d'exécution aux instants $3k+1$ et $3k+2$ pour $k \in \mathbb{N}$, et elle est simultanément active et au repos aux instants $3k$.

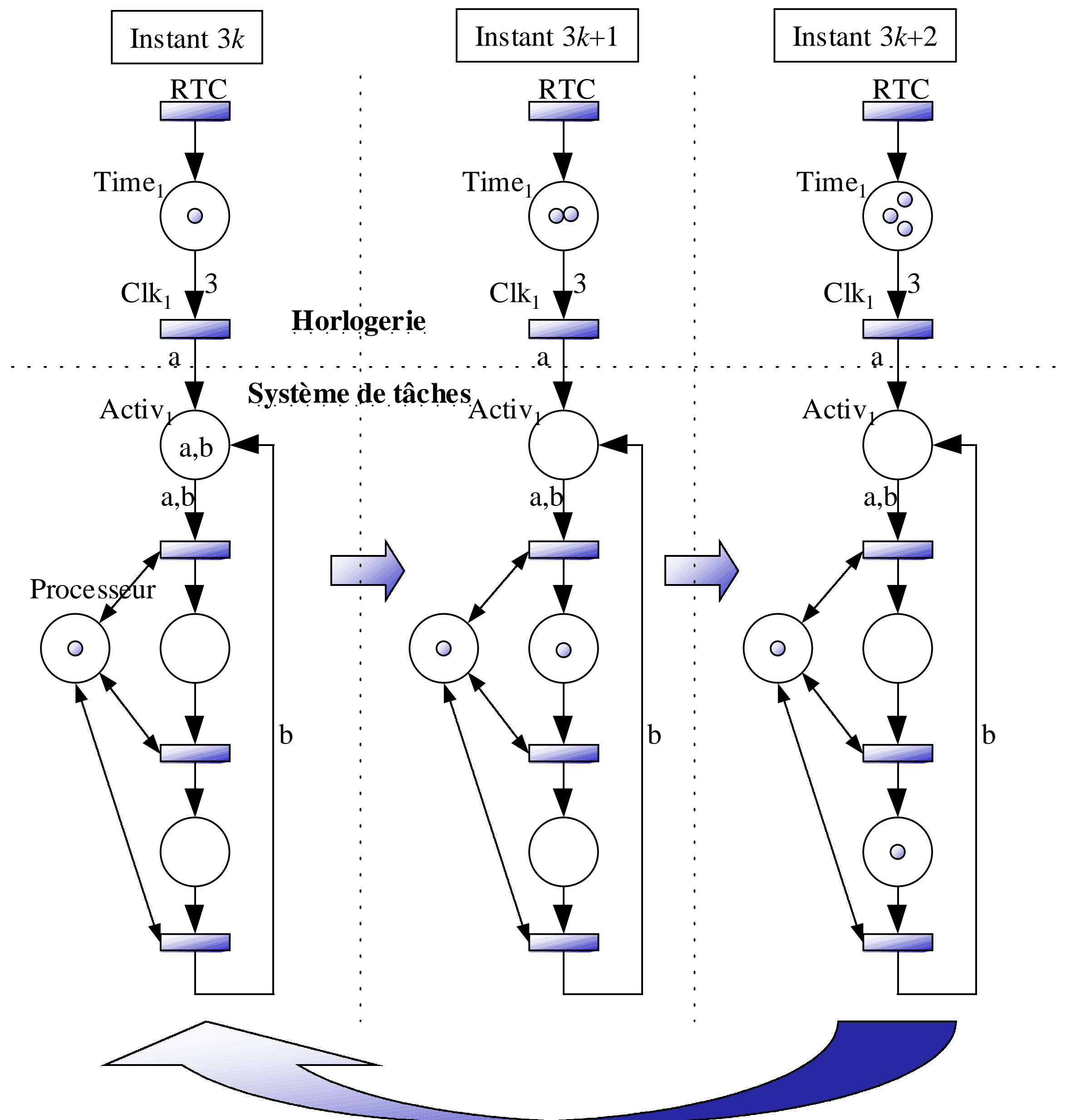


figure IV-1-10 : états du RdP modélisant un système d'une tâche indépendante S={<0,3,3,3>}

Lorsqu'une tâche est en cours d'exécution, une seule des places de la chaîne de transitions contient un ou des jetons, lorsqu'elle n'est pas en cours d'exécution, $activ_i$ contient un jeton *b* produit par sa dernière transition et les autres places de la chaîne de transitions sont vides. Initialement, un jeton *b* existe dans $activ_i$, signifiant qu'elle n'est pas en cours d'exécution. Afin d'éviter toute réentrance, la première transition du système de tâches ne peut être tirée que lorsque la tâche n'est pas en cours d'exécution (i.e. lorsqu'il y a un jeton *b* dans $activ_i$). On dira donc que τ_i est **au repos** lorsque $\{b\} \subseteq M(activ_i)$, notons qu'être au repos ne signifie pas ne pas être active.

Jusqu'ici nous avons présenté comment modéliser la périodicité des tâches, ainsi que les PTR qui les composent. Nous intégrons maintenant les autres caractéristiques temporelles, à savoir, les dates de réveil des tâches, et les contraintes temporelles.

IV-1.5 Prise en compte des temps creux

Comme nous l'avons vu en section I-2.2.3.d, il est important que les techniques d'ordonnancement puissent fournir des séquences non conservatives (i.e. les temps creux doivent pouvoir être placés alors que d'autres tâches peuvent être traitées). Pour permettre aux temps creux d'intervenir n'importe quand, nous modélisons les temps creux cycliques par une tâche oisive τ_0<t_c+1,P(1−U_S),P,P> comme cela est expliqué en section III-2. Cette tâche est une tâche au même titre que les autres tâches d'un système, ce qui permet d'injecter des temps creux cycliques dans les séquences produites par le RdP. De la même façon, il est primordial de pouvoir placer les temps creux acycliques de façon non conservative. A cet effet, nous calculons avant la construction du RdP le nombre de temps creux acycliques présents dans toute séquence d'ordonnancement à l'aide d'un diagramme des charges. Soit n_c ce nombre. Si $n_c>0$, nous adjoignons au RdP une place P_c contenant initialement n_c jetons, reliée à une transition T_c utilisant le processeur comme les transitions des tâches. La transition T_c modélise l'occurrence des temps creux acycliques (voir figure IV-1-11). Cette transition est étiquetée par τ_0 puisque comme la tâche oisive, elle modélise l'occurrence de temps creux. Tous les temps creux sont alors pris en compte soit par l'occurrence d'une transition de τ_0, soit par l'occurrence de T_c. Dans tous les cas, cela se traduit sur le langage du RdP par la lettre τ_0.

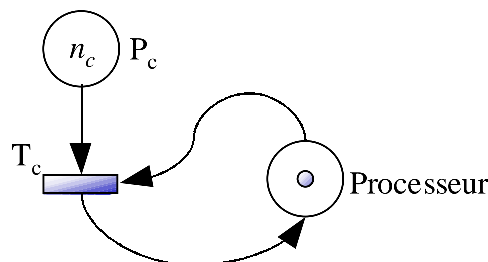


figure IV-1-11 : prise en compte des temps creux acycliques

IV-1.6 Prise en compte des dates de réveil

Les premières dates de réveil des tâches sont prises en compte par le marquage initial du RdP.

- Une tâche τ_i telle que $r_i=0$ est active dès le début de l'application, il faut donc que sa place d'activation contienne un jeton a et un jeton b. Dans ce cas, $M_0(activ_i)=\{a,b\}$, et $M_0(time_i)=1$.
- Une tâche τ_i à réveil différé (i.e. $r_i>0$) doit être retardée de r_i unités de temps avant d'être activée, et n'est initialement pas active et au repos, donc $M_0(Time_i)=P_i-r_i+1$ et $M_0(activ_i)=\{b\}$. La tâche τ_i sera donc activée au bout de r_i unités de temps. Cependant, ce marquage initial n'est valide que dans l'hypothèse où $r_i\leq P_i+1$. Afin d'étendre le modèle aux cas où $r_i>P_i+1$, on ajoute dans ce cas une place $Wait_i$, contenant initialement r_i-P_i+1 jetons, connectée à une transition $Waiting_i$ dont la finalité est de consommer les r_i-P_i+1 premiers jetons produits par RTC dans $Time_i$. Le détail de ce système d'attente initiale est donné sur la figure IV-1-12. D'où $M_0(time_i)=\begin{cases} 0 \text{ si } r_i > P_i+1 \\ Pi - ri + 1 \text{ sinon} \end{cases}$.

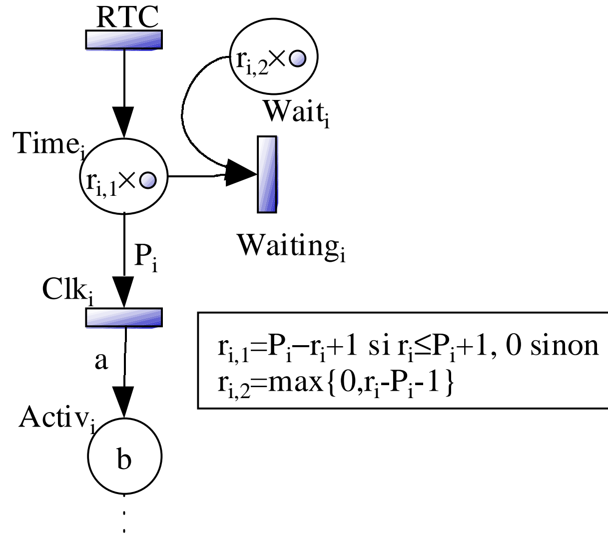


figure IV-1-12 : modélisation de l'horlogerie d'une tâche retardée au-delà de sa période

On ne représente la place $Wait_i$ et la transition $Waiting_i$ que lorsque $r_i>P_i+1$. En effet, dans le cas contraire, la transition $Wait_i$ est morte et est donc inutile. Il est à noter que par hypothèse, $P_i>1$, ce qui implique que les transitions $Waiting_i$ et Clk_i ne sont jamais en conflit : en effet tant que $Wait_i$ contient des jetons, dès que $Time_i$ reçoit un jeton, il est consommé par $Waiting_i$, la place $Time_i$ ne contient donc jamais P_i jetons avant que $Wait_i$ ne soit vide.

IV-1.7 Prise en compte des contraintes temporelles

Les sections précédentes montrent comment modéliser le corps des tâches, avec leur activité concurrente, ainsi que le fait que leurs activations sont périodiques. Il reste un élément majeur à prendre en compte : les échéances des tâches.

IV-1.7.1 Tâches à échéance avant requête

Lorsqu'une tâche τ_i n'est pas à échéance sur requête (i.e. $D_i<P_i$), la place $Time_i$ du RdP contient D_i+1 jetons à l'instant t dans deux cas :

- Soit la tâche n'a pas encore été activée pour la première fois (i.e. la tâche sera activée pour la première fois dans P_i-D_i unités de temps) : dans ce cas, la place $activ_i$ contient un jeton b, déposé dans cette place lors du marquage initial, et pas encore retiré puisque jamais un jeton a n'y a été produit, impliquant que la transition $T_{i,1,1}$ n'a jamais tiré.
- Soit la tâche a déjà été active. Dans ce cas, sa dernière activation a eu lieu à l'instant $t-D_i$ et son échéance vient de tomber. Il faut pour garantir le respect de celle-ci que la tâche ait été exécutée, c'est à dire que le couple $\{a,b\}$ de jetons présents à la date de la dernière activation de τ_i dans $activ_i$ ait été consommé par $T_{i,1,1}$ et que la dernière transition de τ_i ait été tirée, produisant un jeton b dans $activ_i$. La tâche τ_i a donc respecté son échéance si et seulement si le marquage courant $M(activ_i)=\{b\}$.

La figure IV-1-13 illustre le comportement du corps d'une tâche : la première transition nécessite $\{a,b\}$ jetons dans $activ_i$ et la dernière transition produit $\{b\}$ dans $activ_i$. Afin d'être le plus général possible, cette figure comporte aussi le schéma du dispositif d'attente au cas où $r_i>P_i+1$.

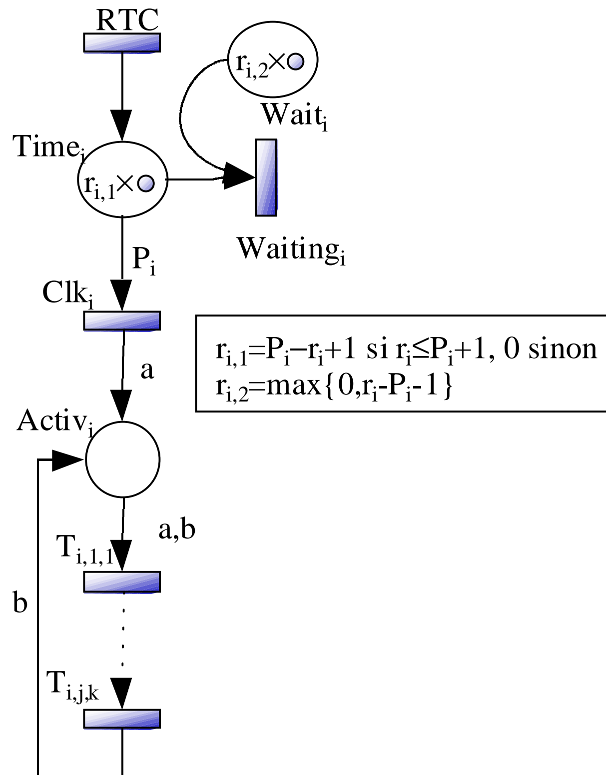


figure IV-1-13 : schématisation du comportement temporel d'une tâche

IV-1.7.2 Tâches à échéance sur requête

Le cas des tâches à échéance sur requête (i.e. $D_i=P_i$) est un cas particulier du cas précédent, mais il est à étudier car étant donné que la transition Clk_i est tirée dès que $Time_i$ contient D_i jetons, le dépassement d'échéance d'une tâche ne peut se représenter de la même façon que dans le cas où $D_i<P_i$. En effet, lorsque l'échéance d'une tâche est atteinte, elle est réactivée. Cela implique qu'un jeton a est produit dans la place $activ_i$, activant la tâche. A cet instant, soit la tâche a terminé sa dernière activation et a donc respecté son échéance, dans ce cas, un jeton b est présent dans $activ_i$, on a donc dans ce cas $M(activ_i)=\{a,b\}$. Par contre, si la tâche n'a pas respecté son échéance, alors $activ_i$ ne contient qu'un jeton $\{a\}$ ou bien encore $\{a,b\}$, ce qui signifie qu'elle n'a même pas commencé son exécution. De façon générale, une tâche non terminée ne doit pas être activée (c'est l'hypothèse de non réentrance des tâches avec $D_i \le P_i$) donc pour s'assurer qu'une tâche à échéance sur requête respecte son échéance, il faut s'assurer que lorsque $M(Time_i)=1$, $M(activ_i)=\{a,b\}$ ou $\{b\}$ (ce dernier cas correspond au cas particulier où la tâche n'a jamais encore été activée).

IV-1.7.3 Cas général

Dans les deux cas, la place $activ_i$ doit contenir un jeton b lorsque l'horloge locale contient plus de D_i jetons, et la place $activ_i$ ne doit jamais contenir uniquement un jeton a. Etant donné qu'on veut ne s'intéresser qu'aux comportements valides des tâches, le RdP se voit associer les contraintes terminales suivantes sur les marquages :

$(M(Time_i)=1 \Rightarrow M(Activ_i)=\{a,b\}$ ou $\{b\}) \wedge (M(Time_i)=D_i+1 \Rightarrow M(Activ_i)=\{b\})$

Rappelons que notre objectif est de construire l'ensemble des séquences d'ordonnancement valides du système de tâches. Nous définissons donc un alphabet où une lettre est le nom d'une tâche : soit $\Sigma_S=\{\tau_0,\ldots,\tau_n\}$ cet alphabet, et $\sigma_S : T \rightarrow \Sigma_S$ la fonction d'étiquetage définie

par $\forall T_{i,j,k} \in T$, $\sigma_S(T_{i,j,k})=\tau_i$, et $\sigma_S(t)=\varepsilon$ pour toute autre transition $t \in T$. Cet étiquetage a pour effet d'ignorer les transitions du système d'horlogerie, et de ne prendre en compte que les transitions du système de tâches, en ne les différenciant que par leur appartenance à une tâche.

Un mot du langage du RdP ainsi étiqueté est une séquence d'ordonnancement du système de tâches modélisé. Comme nous ne nous intéressons qu'aux séquences d'ordonnancement valides du système, nous nous limitons à l'étude du langage qui n'est formé que de mots dont chacun des états respecte les contraintes terminales assurant la validité temporelle des tâches. Nous avons vu qu'un tel sous-ensemble du langage terminal, dont chaque état respecte les contraintes terminales du RdP, était appelé centre du langage terminal.

Afin de s'assurer que les mots du langage considéré ne contiennent que des séquences d'ordonnancement valides, nous définissons donc l'ensemble des marquages terminaux $\Psi=\{M \in (\mathbb{N} \times \{a\} \times \{b\})^{|Q|} \mid (M(time_i)=1 \Rightarrow M(activ_i)=\{a,b\}$ ou $\{b\}) \wedge (M(time_i)=D_i+1 \Rightarrow M(activ_i)=\{b\})\}_{i=0..n}$. Avec cet ensemble de contraintes, le centre du langage terminal est exactement l'ensemble de toutes les séquences d'ordonnancement valides du système de tâches modélisé.

IV-1.8 Récapitulatif

La définition suivante formalise le modèle présenté dans les paragraphes précédents afin de montrer la correction du modèle par rapport aux systèmes de tâches à modéliser.

Définition IV-1-1 : Soit $S=\{\tau_i=\langle r_i,C_i,D_i,P_i\rangle\}_{i=0..n}$ un système de tâches périodiques définies par un pseudo code constitué d'une série d'instructions temps réel (instructions indépendantes abstraites à leur durée, envoi et réception de messages par boîtes aux lettres, accès en lecture ou écriture à une ressource), et soient $\Re=\{R_1, \ldots,R_{|\Re|}\}$ les ressources critiques du système accessibles uniquement en écriture, $\Im=\{F_1,\ldots,F_{|\Im|}\}$ les ressources critiques accessibles en lecture ou écriture. On note $|R_i|$ le nombre d'exemplaires de la ressource R_i et $|F_i|$ le nombre maximal de lectures simultanées de la ressource F_i. Enfin soient $\wp=\{B_1,\ldots,B_{|\wp|}\}$ les boîtes aux lettres utilisées par les tâches.

Le modèle réseau de Petri de S est défini par $R=\langle N,M_0\rangle$, avec $N=\langle Q,T,F,W\rangle$ où :

- $T = \{RTC, Clk_i, Waiting_i, T_{i,j,k}, T_c\}_{i\in 0..n, j\in\{1..Ci\}, k\in\{1..3\}}$,
- $Q=\{Processeur,Time_i,Wait_i,Activ_i,P_{i,j,k},P_c,R_r,F_f,B_b\}_{i\in 0..n,j\in\{1..Ci\},k\in\{1..3\},r\in\{1..|\Re|\},f\in\{1..|\Im|\},b\in\{1..|\wp|\}}$,
- F et W sont définies comme décrit dans les sections précédentes,
- M_0 est défini par :

✓ $M_0(Processeur)=1$,

✓ $\forall i\in 0..n$, $M_0(Time_i)= \begin{cases} 0 \text{ si } r_i > P_i+1 \\ Pi-ri+1 \text{ sinon} \end{cases}$,

✓ $\forall i\in 0..n$, $M_0(Wait_i)=max\{0, r_i-P_i-1\}$,

- ✓ $\forall i \in 0..n$, $M_0(Activ_i)=\{b,a^{(P_i-r_i)\text{ div }P_i}\}=\begin{cases}\{a,b\} \text{si } r_i = 0 \\ \{b\} \text{si } r_i > 0\end{cases}$,
- ✓ $\forall i \in 1..|\Re|$, $M_0(R_i)=|R_i|$,
- ✓ $\forall i \in 1..|\Im|$, $M_0(F_i)=|F_i|$,
- ✓ $\forall i \in 1..|\wp|$, $M_0(B_i)=0$,
- ✓ $\forall i \in 0..n, j \subseteq \{1..Ci\}, k \subseteq \{1..3\}$, $M_0(P_{i,j,k})=0$

Le centre du langage terminal $L(R,\sigma,\Psi)$ du RdP coloré avec la règle de tir maximal décrit ci-dessus est l'ensemble de toutes les séquences d'ordonnancement valides du système de tâches modélisé, avec :

$\Psi=\{M \subseteq (\mathbb{N} \times \{a\} \times \{b\})^{|Q|} | (M(time_i)=1 \Rightarrow M(activ_i)=\{a,b\}$ ou $\{b\}) \wedge$
$(M(time_i)=D_i+1 \Rightarrow M(activ_i)=\{b\})\}_{i=0..n}$.

En effet, par construction, il existe un isomorphisme entre le système de tâches modélisé et le modèle RdP le modélisant. La preuve est donnée dans [Gro 96] pour le cas où les tâches sont simultanées, où les tâches communicantes ont même période, où les ressources sont binaires (pas de multi-instances ni de ressources en lecture-écriture), et où les tâches sont préemptibles. Elle peut être étendue dans le cas plus général considéré de la façon suivante : la propriété étudiée est l'ensemble des ordonnancements valides d'un système de tâches. Un système de tâches est donné par le pseudo-code constituant chaque tâche. Il est composé d'une part de paramètres temporels, et d'autre part de blocs représentants des durées, et de primitives de communication, d'accès à des ressources, et de préemptibilité mises en séquence. Chacune de ces instructions est un constructeur du pseudo-code, et à chaque constructeur correspond un motif du modèle RdP. On peut montrer que les propriétés de chaque motif du modèle vérifient les propriétés de chaque constructeur du pseudo-code donné, et que les propriétés de mise en séquence (pour le RdP) des motifs du modèle vérifient les mêmes propriétés que la mise en séquence des constructeurs.

En fait, le RdP peut être vu comme une modélisation ou une implémentation du système de tâches, prenant en compte ses caractéristiques structurelles et temporelles. Son langage terminal est donc l'ensemble de tous les comportements possibles du système de tâches. Cependant, nous nous intéressons à la production de séquences d'ordonnancement valides, c'est à dire au centre du langage terminal : les contraintes terminales excluent tout état du réseau pour lequel une échéance serait violée.

Remarque : les séquences d'ordonnancement obtenues contiennent non seulement les séquences produites par les algorithmes d'ordonnancement classiques tels *ED*, *DM*,… , les séquences d'ordonnancement conservatives (c'est à dire au plus tôt), mais aussi toutes les autres séquences dans lesquelles le processeur peut rester oisif alors qu'il y a des tâches actives et non terminées. Cela est essentiel afin de s'assurer de trouver au moins une séquence s'il en existe une, car comme nous l'avons vu dans le Théorème I-2-29, les ordonnancements conservatifs ne sont pas optimaux dès lors que des ressources critiques sont en jeu. L'obtention des séquences non conservatives est assurée par l'utilisation de la tâche oisive qui modélise les temps creux cycliques et de la transition T_c qui modélise les temps creux acycliques : on peut donc choisir arbitrairement de ne rien faire alors que des tâches pourraient être exécutées.

IV-1.9 Exemples de modélisation

IV-1.9.1 Tâches à échéance sur requête à départ simultané, avec une ressource en écriture et une boîte aux lettres

Afin d'illustrer le modèle proposé, la figure IV-1-14 représente le modèle du système de tâches donné dans l'exemple IV-1-4.

exemple IV-1-4 : un système de 3 tâches communicantes

```
Tâche τ1 est
r1 = 0 ;
P1 = 8 ;
D1 = 8 ;
Début
  b1,1=2 ;
  Déposer_BAL(MB) ;
  b1,2=1 ;
Fin ;

Tâche τ2 est
r2 = 0 ;
P2 = 8 ;
D2 = 8 ;
Début
  Retirer_BAL(MB) ;
  b2,1=1 ;
  Prendre_Sémaphore(R,1) ;
  b2,2=1 ;
  Rendre_Sémaphore(R,1) ;
Fin ;

Tâche τ3 est
r3 = 0 ;
P3 = 16 ;
D3 = 16 ;
Début
  b3,1=1 ;
  Prendre_Sémaphore(R,1) ;
  b3,2=2 ;
  Rendre_Sémaphore(R,1) ;
  b3,3=1 ;
Fin ;

Tâche τ0 est
r0 = 0 ;
P0=ppcm(8,8,16)=16 ;
D0=P0=16 ;
Début
  b0,1=16×(1-(3/8+2/8+4/16))=2 ;
Fin ;
```

Les marquages terminaux sont donnés par $\Psi=\{M\in(\mathbb{N}\times\{a\}\times\{b\})^{|Q|} \mid (M(time_i)=1\Rightarrow M(activ_i)=\{a,b\} \text{ ou } \{b\})\}_{i=0..3}$. On se contente de la première partie des contraintes car dans cet exemple, toutes les tâches sont à échéance sur requête. De plus, puisque toutes les dates de réveil vérifient $r_i\leq P_i+1$, il n'est pas nécessaire de créer les places $wait_i$ et les transitions associées $waiting_i$. Afin de simplifier les représentations graphiques des modèles de tâches, nous omettons les arcs reliant chaque transition de la partie système de tâches à la ressource processeur. Les cadres entourant les transitions d'horlogerie et du système de tâches sont représentés dans le but de rappeler que toute transition du cadre système de tâches est reliée au processeur.

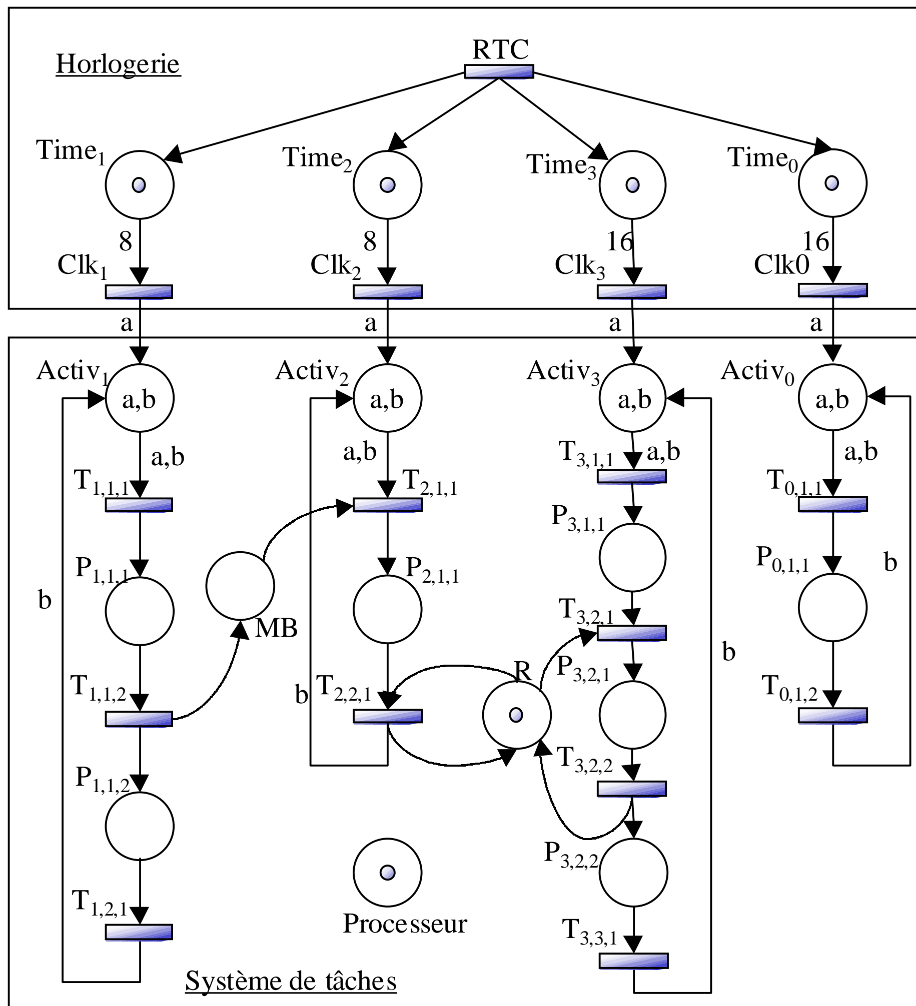


figure IV-1-14 : modélisation d'un système de tâches interagissantes à l'aide d'un RdP, on ne représente pas $W(Processeur,T_{i,j,k})=W(T_{i,j,k},Processeur)=1$

IV-1.9.2 Illustration de l'obtention des séquences non conservatives

Afin d'illustrer l'intérêt de la tâche oisive, nous comparons une modélisation avec et une modélisation sans tâche oisive sur un exemple de deux tâches non préemptibles mutuellement (i.e. partageant une ressource critique qui les empêche de se préempter). Cet exemple a été utilisé en section I-2.2.3.d afin de montrer que les ordonnancements au plus tôt n'étaient pas dominants, et est redonné dans l'exemple IV-1-5.

---

exemple IV-1-5 : un système de tâches non ordonnançable au plus tôt.

Tâche τ_1 est
r_1 = 0 ;
P_1 = 4 ;
D_1 = 4 ;
Début
  Prendre_Sémaphore(R,1) ;
  $b_{1,1}$=2 ;
  Rendre_Sémaphore(R,1) ;
Fin ;

Tâche τ_2 est
r_2 = 0 ;
P_2 = 5 ;
D_2 = 1 ;
Début
  Prendre_Sémaphore(R,1) ;
  $b_{2,1}$=1 ;
  Rendre_Sémaphore(R,1) ;
Fin ;

Le modèle R de ce système, illustré sur la figure IV-1-15(a), ne donne aucune séquence d'ordonnancement : le centre de son langage $L(R,\sigma,\Psi)$, avec $\Psi=\{M\in(\mathbb{N}\times\{a\}\times\{b\})^{|Q|} \mid (M(time_1)=1\Rightarrow M(activ_1)=\{a,b\}$ ou $\{b\}) \wedge (M(time_2)=2\Rightarrow M(activ_2)=\{b\})\}$, est vide car le fonctionnement du RdP pris en compte est la règle de tir maximal.

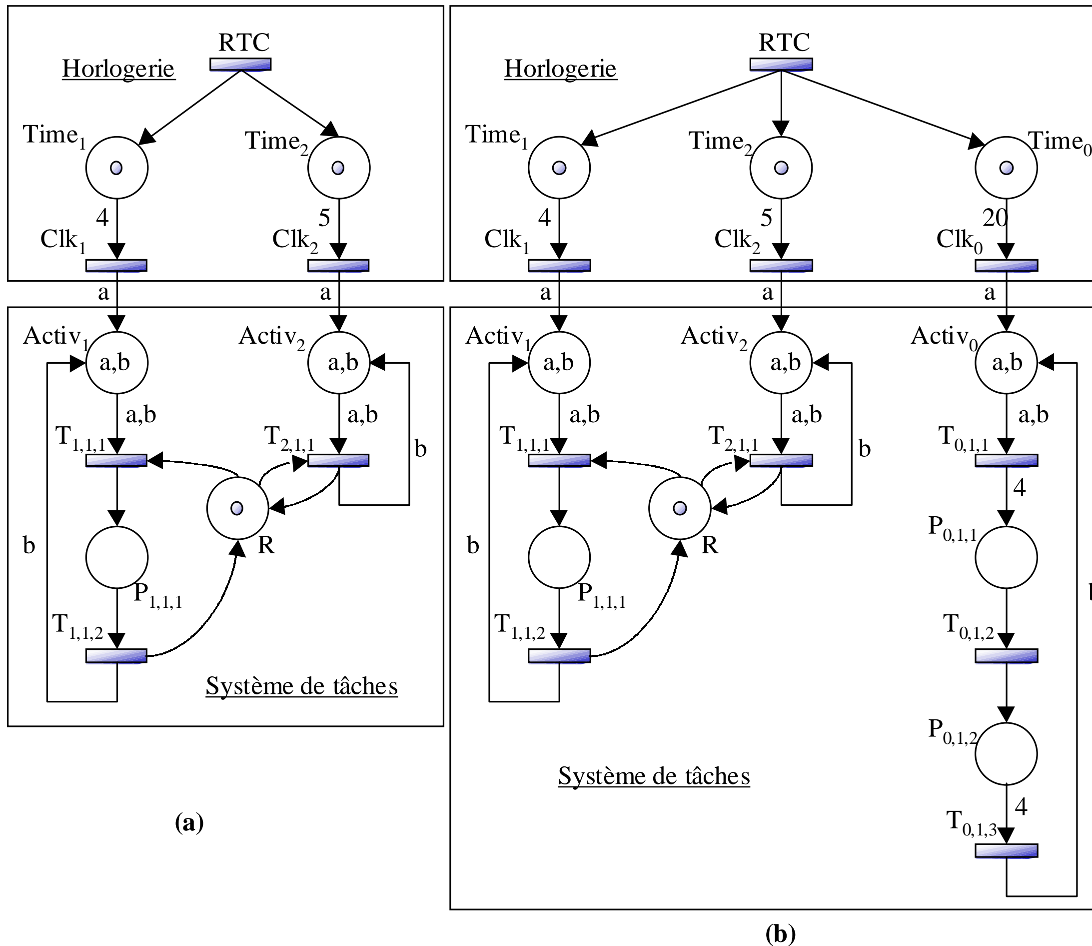


figure IV-1-15 : (a) un modèle réseau de Petri sans tâche oisive, (b) le même système avec tâche oisive

On n'obtient donc que les ordonnancements conservatifs du système modélisé car on ne peut pas ne pas faire tirer une transition franchissable, ce qui implique que si une tâche du système peut progresser (cela correspond au fait que la transition correspondante est franchissable), alors au moins l'une des transitions du système de tâches est tirée. Etant donné que le nombre des temps creux présents lors de chacune des métapériodes est connu, il suffit de modéliser la possibilité pour ces temps creux d'apparaître à n'importe quel instant afin que le langage du RdP ne soit pas limité aux séquences d'ordonnancement au plus tôt. Le modèle obtenu, représenté sur la figure IV-1-15(b), produit 54 séquences d'ordonnancement.

IV-1.9.3 Tâches comportant des parties non préemptibles

Il peut être intéressant de modéliser des tâches dont certaines sont non préemptibles par les autres. En effet, dans certaines applications temps réel, certaines tâches accédant à du matériel sensible ou à des vecteurs d'interruptions se protègent contre toute préemption en masquant les interruptions. Cela peut être fait en utilisant des instructions en assembleur comme *CLI*

(*CLear Interrupts*) pour masquer et *STI* (*SeT Interrupts*) pour démasquer les interruptions. Cela est assez dangereux algorithmiquement mais assurer l'atomicité de certaines actions peut être nécessaire. Notre modélisation permet de prendre en compte simplement ce besoin : il suffit de modéliser la partie non préemptible d'une tâche par des transitions qui ne rendent pas le processeur après leur tir. Dans ce but, la première transition $T_{i,j,1}$ suivant une instruction *CLI* est telle que $W^{-}(Processeur,T_{i,j,k})=1$ mais $W^{+}(T_{i,j,k},Processeur)=0$. Au moment où $T_{i,j,k}$ est tirée, la tâche τ_i ne rend pas le processeur, qui n'est alors plus disponible pour aucune tâche. Les transitions $T_{i,k,m}$ modélisant les blocs compris entre *CLI* et *STI* sont créées telles que $W^{-}(Processeur,T_{i,k,m})=W^{+}(T_{i,k,m},Processeur)=0$, sauf la dernière transition $T_{i,p,q}$ précédent *STI* qui rend le processeur disponible aux autres tâches, donc $W^{-}(Processeur,T_{i,p,q})=0$ et $W^{+}(T_{i,p,q},Processeur)=1$.

L'exemple exemple IV-1-6 illustre l'utilisation du modèle afin de rendre une partie de tâche non préemptible.

---

exemple IV-1-6 : 3 tâches, τ_1 est non préemptible, τ_3 est différée et τ_2 est différée de plus d'une période.

Tâche τ1 est	Tâche τ2 est	Tâche τ3 est
r1 = 0 ;	r2 = 15 ;	r3 = 2 ;
P1 = 4 ;	P2 = 8 ;	P3 = 8 ;
D1 = 4 ;	D2 = 4 ;	D3 = 5 ;
Début	Début	Début
CLI ;	b2,1=1 ;	b3,1=3 ;
b1,1=2 ;	Fin ;	Fin ;
STI ;		
Fin ;		

Ce système a une charge U=1, il n'y a donc pas de temps creux cycliques, donc pas de tâche oisive. Cependant, étant donné que les tâches sont différées, il nous faut construire le diagramme des charges afin de connaître le nombre de temps creux acycliques apparaissant dans les séquences. Ce diagramme montre qu'il y a un temps creux acyclique et qu'il apparaît à l'instant *7*. Nous en déduisons la profondeur du graphe des marquages, qui est donc de $7+1+8=16$, et le marquage initial de la place P_c : $M_0(P_c)=1$.

La figure IV-1-16 représente le modèle RdP pour l'exemple IV-1-6. Les marquages terminaux sont donnés par :

$\Psi=\{M\in(\mathbb{N}\times\{a\}\times\{b\})^{|Q|} \mid (M(time_1)=1\Rightarrow M(activ_1)=\{a,b\}$ ou $\{b\}) \wedge (M(time_2)=5\Rightarrow M(activ_2)=\{b\})\wedge(M(time_3)=6\Rightarrow M(activ_3)=\{b\})\}$.

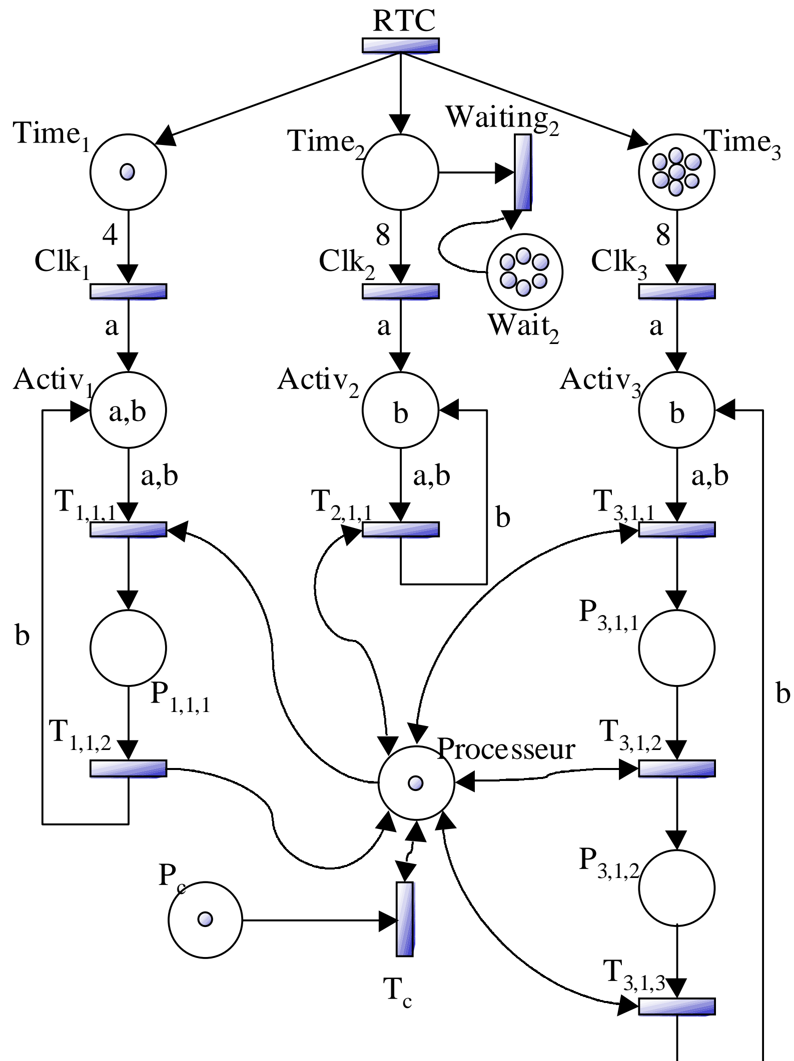


figure IV-1-16 : modélisation d'un système de trois tâches : τ_1 est non préemptible, les deux autres différées

IV-2 Etude

La méthodologie d'obtention de séquences d'ordonnancement valides repose sur l'obtention du centre du langage terminal du modèle RdP. En effet chaque mot de longueur L du langage terminal est une séquence d'ordonnancement valide de longueur L. Cependant, nous nous intéressons aux séquences de longueur infinie, donc nous ne voulons que les mots de longueur infinie. Ceux-ci sont obtenus par la construction d'un graphe des marquages particulier : un graphe ne contenant que des mots de longueur infinie. Il convient donc d'étudier ce graphe, ainsi que sa taille qui n'est pas polynomiale.

IV-2.1 Profondeur du graphe d'accessibilité

Etant donné que l'ensemble des séquences d'ordonnancement est l'ensemble des mots infinis du centre du langage du RdP, chaque séquence d'ordonnancement valide du système modélisé est un chemin du graphe d'accessibilité du RdP dans lequel on supprime tout état ne faisant pas partie d'un mot infini.

Rappelons que les arcs du graphe sont étiquetés par $\{\tau_{i, i=0..n}\}^{|T|}\cup\{\varepsilon\}$, en effet, au plus $|T|$ transitions sont tirées à chaque instant puisque dans le modèle, aucune transition ne peut être réentrante.

De plus, en mode monoprocesseur, une seule transition du système de tâches peut être tirée à chaque instant à cause de la ressource processeur. Cela implique qu'un arc est étiqueté par $\{\tau_{i, i=0..n}\}\cup\{\varepsilon\}$: un arc étiqueté par $\{\tau_i\}$ représente le franchissement d'un ensemble de transition contenant une transition de τ_i, de par l'utilisation du processeur comme une ressource commune aux transitions étiquetées par un nom de tâches, les autres transtions sont nécessairement étiquetées par le mot vide. Puisque les arcs sont étiquetés par un singleton, nous omettrons par la suite les accolades. Un arc étiqueté par ε représente une unité de temps pendant laquelle aucune tâche n'a progressé, or les systèmes modélisés se voient toujours augmenter d'une tâche oisive afin d'atteindre *100%* de charge, et d'une transition modélisant les temps creux acycliques. Cela implique qu'à tout instant, au moins une transition du système de tâches ou bien la transition T_c est franchie. Chaque arc du graphe des marquages est donc étiqueté par une lettre de l'alphabet $\{\tau_{i, i=0..n}\}$. La profondeur du graphe est donnée par une étude préalable du diagramme de charge du système de tâches modélisé. Nous obtenons ainsi la date de réveil de la tâche oisive, le marquage initial de la place P_c et la date maximale d'occurrence du dernier temps creux acyclique t_c. Etant donné que toute séquence d'ordonnancement est périodique de période P après le dernier temps creux acyclique, nous déduisons la profondeur du graphe des marquages.

Théorème IV-2-1 : La profondeur du graphe des marquages d'un modèle réseau de Petri d'un système de tâches est t_c+P+1 avec :

- t_c date du dernier temps creux acyclique, de plus $t_c<r+P$,
- P est le plus petit commun multiple des périodes des tâches du système.

Preuve : découle directement du Théorème III-1-3.
❑

IV-2.2 Représentation du graphe d'accessibilité

Afin d'illustrer l'allure qu'ont les graphes d'accessibilité du modèle, ce qui servira à expliquer comment sont extraites des séquences d'ordonnancement, nous présentons quelques exemples de graphes d'accessibilité.

IV-2.2.1 Deux tâches simultanées indépendantes de charge 100%

Reprenons l'exemple IV-1-3 présentant deux tâches indépendantes simultanées τ_1<0,2,4,4> et τ_2<0,1,2,2>. Leur charge est de *100%*, il n'est donc pas nécessaire d'adjoindre au système une tâche oisive. Les ordonnancements valides de ce système sont l'ensemble des chemins du graphe d'accessibilité, représenté sur la figure IV-2-1, obtenu à l'aide du modèle.

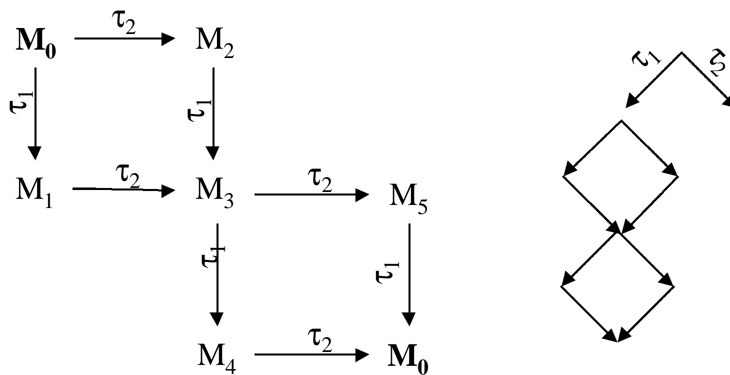


figure IV-2-1 : deux visualisations d'un graphe d'accessibilité contenant l'ensemble des séquences d'ordonnancement valides d'un système de tâches

Le graphe des marquages (noté GM) de gauche exprime le graphe obtenu avec une numérotation des marquages M_i telle qu'elle serait obtenue en faisant une construction en largeur. M_0 est un marquage d'accueil par lequel toute séquence d'ordonnancement passe à la profondeur kP pour $k \in \mathbb{N}$. Une séquence d'ordonnancement valide du système est $\rho=(\tau_1, \tau_2, \tau_2, \tau_1, \tau_2, \tau_2)^*$ puisque c'est un mot de longueur infinie du centre du langage. Sur la partie droite de la figure IV-2-1, une représentation plus concise du même graphe des marquages est donnée dans laquelle on abstrait les marquages qui n'ont que peu d'intérêt pour l'étude des séquences d'ordonnancement. Dans le cas des systèmes de tâches simultanées, il n'y a pas de temps creux acyclique et chaque séquence d'ordonnancement est de longueur *P*. Dans ce cas, le GM est de la forme d'un **diamant**[11] (car tout chemin est permutation d'un autre), dans lequel le nœud initial et le nœud final sont M_0. Soit *W* l'ensemble des mots de longueur *P* étiquetant un chemin partant de M_0 (et donc allant à M_0), alors n'importe quelle combinaison infinie de la concaténation de mots *W* est un ordonnancement valide. En particulier, si $w \in W$, alors w^* est un ordonnancement valide du système.

IV-2.2.2 Deux tâches différées indépendantes de charge 100%

Nous nous intéressons dans ce paragraphe au cas général de tâches différées. L'exemple IV-2-1 illustre un graphe d'ordonnancement à tâches différées typique, à travers le GM du modèle réseau de Petri associé donné sur la figure IV-2-2.

[11] Un graphe est un diamant si il a un nœud source et que tout chemin partant du nœud source mène au même nœud, appelé nœud final.

exemple IV-2-1 : paramètres temporels de deux tâches indépendantes différées de charge *100%.* Leur pseudo code est omis puisqu'il est seulement composé de l'instruction $b_{i,1}=C_i$.
τ_1<5,3,7,7>, τ_2<0,8,14,14>

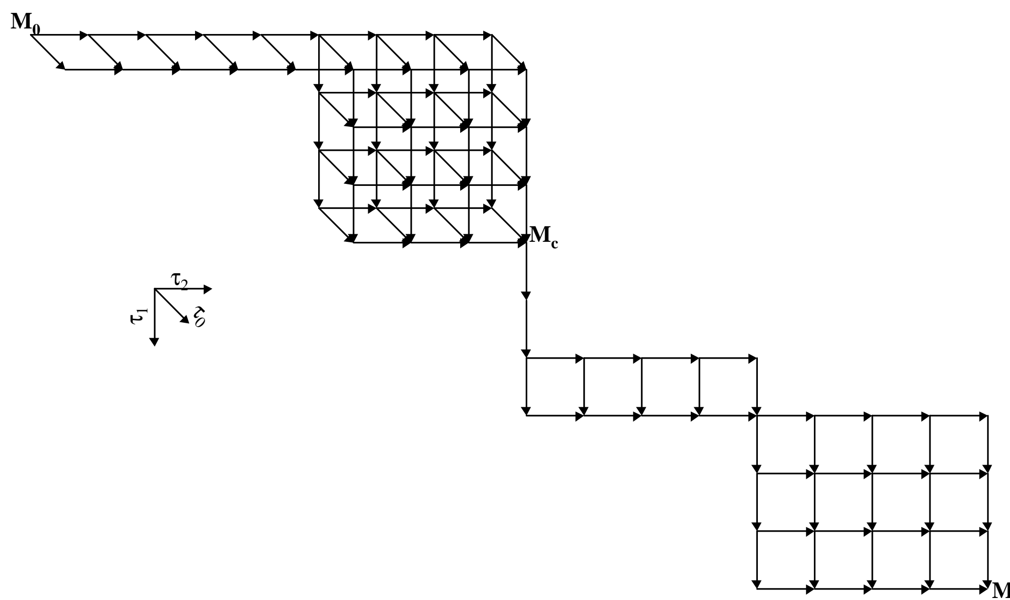


figure IV-2-2 : un graphe d'accessibilité contenant l'ensemble des séquences d'ordonnancement valides d'un système de tâches différées

Le diagramme des charges de ce système indique que t_c=11 et qu'il y a un temps creux acyclique. D'après l'extension du Théorème III-1-3 aux algorithmes non conservatifs, ce temps creux peut intervenir entre les instants *0* et *11*, ensuite le système entre dans son régime permanent : c'est le marquage M_c situé à la hauteur t_c+1 qui marque la sortie du diamant correspondant à la partie transitoire des séquences d'ordonnancement (toute séquence d'ordonnancement passe par ce marquage), et l'entrée du diamant contenant la partie périodique des séquences d'ordonnancement, nous dénotons *diamant*(M_0,M_c) l'ensemble de tous les mots du diamant correspondant à la partie transitoire des séquences et *diamant*(M_c,M_c) les mots du diamant correspondant à la partie périodique des séquences. Tout mot infini formé d'un mot de *diamant*(M_0,M_c) concaténé avec un ensemble de mots de *diamant*(M_c,M_c) est un ordonnancement valide. En particulier, si $w_1 \in$ *diamant*(M_0,M_c) et $w_2 \in$ *diamant*(M_c,M_c), alors $w_1 w_2^*$ est un ordonnancement valide.

IV-2.3 Complexité et taille du graphe d'accessibilité

Puisque le problème de l'ordonnancement de systèmes de tâches temps réel est *NP*-difficile dans le cas général, il est inévitable (sauf si *P=NP*) qu'une démarche exhaustive ne soit pas polynomiale, puisque c'est une démarche optimale dans le sens où si une séquence d'ordonnancement existe alors elle est trouvée.

Nous nous intéressons donc à la taille et à la complexité de l'obtention du graphe des marquages dans le cas général. Celui-ci peut effrayer à juste titre au premier abord. En effet, soient *n* tâches simultanées identiques indépendantes de durée unitaire, à échéance sur requête de période *n*, le nombre séquences d'ordonnancement d'un tel système est donné par la permutation de *n* éléments parmi *n*, c'est à dire *n!* séquences. Le nombre de nœuds d'un graphe

des marquages contenant ces séquences est constitué quant à lui de 2^n nœuds et de l'ordre de $n2^n$ arcs.

De plus, supposons que les périodes des tâches ne soient plus identiques mais premières entre elles afin de faire croître arbitrairement la durée de simulation, la profondeur du graphe n'est pas polynomiale car le *ppcm* de nombres n'est pas borné polynomialement en fonction de ses arguments.

IV-2.3.1 Borne théorique à la taille du graphe d'accessibilité

La profondeur du graphe des marquages est au moins exponentielle, car le nombre de requêtes d'une tâche pendant la durée de simulation est de l'ordre de $P=ppcm(P_j)_{j=1..n}/P_i$ (voir la preuve en annexe I-1). Cependant, on peut difficilement imaginer l'existence d'un système de tâches dans lequel un nombre important de périodes différentes est utilisé, avec des périodes premières entre elles ou bien avec un *pgcd* réduit [Ram 90]. Par exemple, chaque groupe de tâches communicantes est soumis à des contraintes de corrélation des périodes. D'ailleurs, la plupart des auteurs travaillant sur des approches hors-ligne se sont affranchis de cette complexité en considérant le problème donné sous la forme des requêtes des tâches [XP 90]. C'est en partie pour cela que notre approche est, à notre connaissance, la seule approche hors-ligne prenant en compte les tâches périodiques différées.

Parallèlement à cette complexité, inhérente à toute séquence construite hors-ligne, donc qui ne peut être réduite, la complexité liée à l'exhaustivité des séquences d'ordonnancement est à minimiser. Nous nous intéressons à la taille du *GM*. En effet, au pire, chaque tâche $\tau_{i,i=0..n}$ est exécutée P/P_i en P unités de temps et chacune de ses requêtes nécessite au plus C_i tirs de transitions, on peut donc englober le *GM* dans un pavé totalement maillé dans n+1 dimensions (car il y a n tâches plus la tâche oisive). Il s'agit d'un hyperpavé.

Définition IV-2-1 : Un hyperpavé de dimension n de cotes respectives $d_0,\ldots,d_{n-1}$ est un graphe non orienté $G=(V,E)$ défini inductivement sur le nombre de dimensions comme suit :

- Si $n=1$, $V=\{v_0, v_2,\ldots,v_{d_0}\}$, et E est défini uniquement pour les couples (v_{i-1},v_i) pour $i=1..d_0$,
- Pour $n>1$, V est le produit cartésien de n hyperpavés de dimension 1.

Un hyperpavé contient exactement $\prod_{i=0}^{n-1}(d_i+1)$ sommets et au plus (n+1) arcs par sommets.

La figure IV-2-3 représente un hyperpavé de dimension 3, et de cotes respectives {3,4,2}.

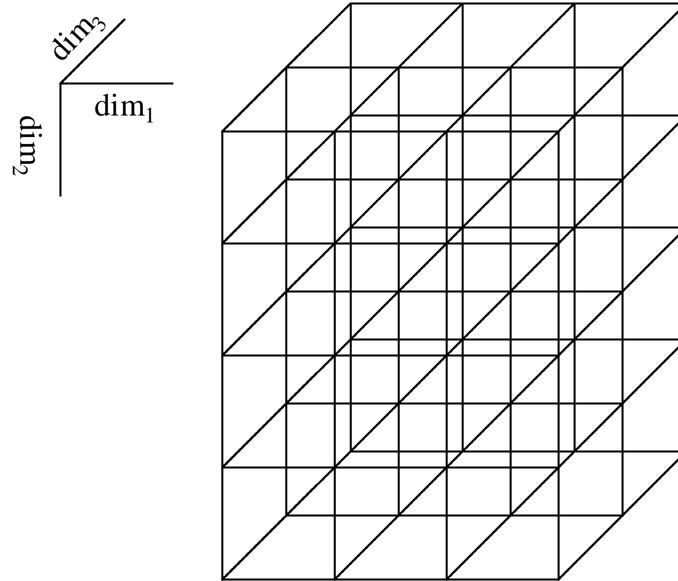


figure IV-2-3 : un hyperpavé de dimension 3 et de cotes {3,4,2}

Proposition IV-2-1 :Un *GM* est totalement contenu dans un hyperpavé de n+1 dimensions *0..n* où les côtes des dimensions *0..n* sont respectivement $\left\lceil \frac{(t_c + P + 1)C_i}{P} \right\rceil_{i=1..n}$ et la côte de la dernière dimension est (C_0+n_c) avec n_c le nombre de temps creux acycliques.

Preuve : A chaque instant, au pire, n+1 (n pour le tâches, plus une pour la tâche oisive ou une pour la tâche creuse) transitions en conflit peuvent être tirées, donnant naissance à au plus n+1 marquages fils par nœud. ❑

Le nombre maximal de nœuds du GM limité à une profondeur P est donc $(1+n_c)\times\prod_{i=0}^{n}\frac{(t_c+P+1)C_i}{P_i}$, étant donné que dans le pire des cas, t_c+P+1 vaut $r+2P$, cela est inférieur ou égal à $(1+n_c)(r+2P)^{n+1}\prod_{i=0}^{n}\frac{C_i}{P_i}$. Etant donné que $\frac{C_i}{P_i}\le 1$, la composante $\prod_{i=0}^{n}\frac{C_i}{P_i}$ diminue le nombre de nœuds, alors que la taille du *ppcm* l'augmente. Cependant, plus ce nombre augmente, plus le nombre de requêtes augmente. Mais étant donné qu'à chaque requête correspond une échéance, le *GM* correspondant aux ordonnancements d'un système de tâches avec plusieurs requêtes par tâche ne peut pas s'étaler totalement sur un hyperpavé.

Les deux exemples suivants illustrent ce fait :

---

exemple IV-2-2 : S_1={τ_1<0,4,7,7>, τ_2<0,6,14,14>} contient deux tâches indépendantes, τ_1 a deux requêtes par *ppcm*, τ_2 n'en a qu'une. Leur charge est maximale, et il n'y a donc pas de tâche oisive.

S_2={τ_1<0,3,7,7>, τ_2<0,9,21,21>} contient deux tâches indépendantes. Cette fois τ_1 a 3 requêtes par *ppcm*.

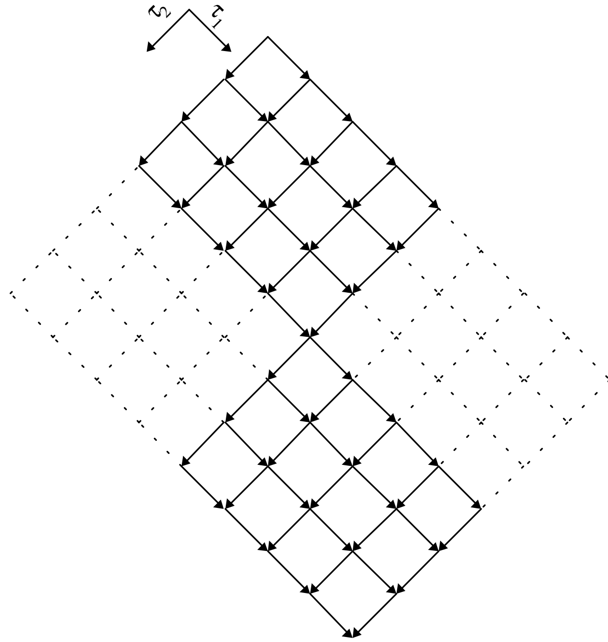


figure IV-2-4 : le GM de S_1 n'occupe que la moitié de l'hyperpavé le bornant

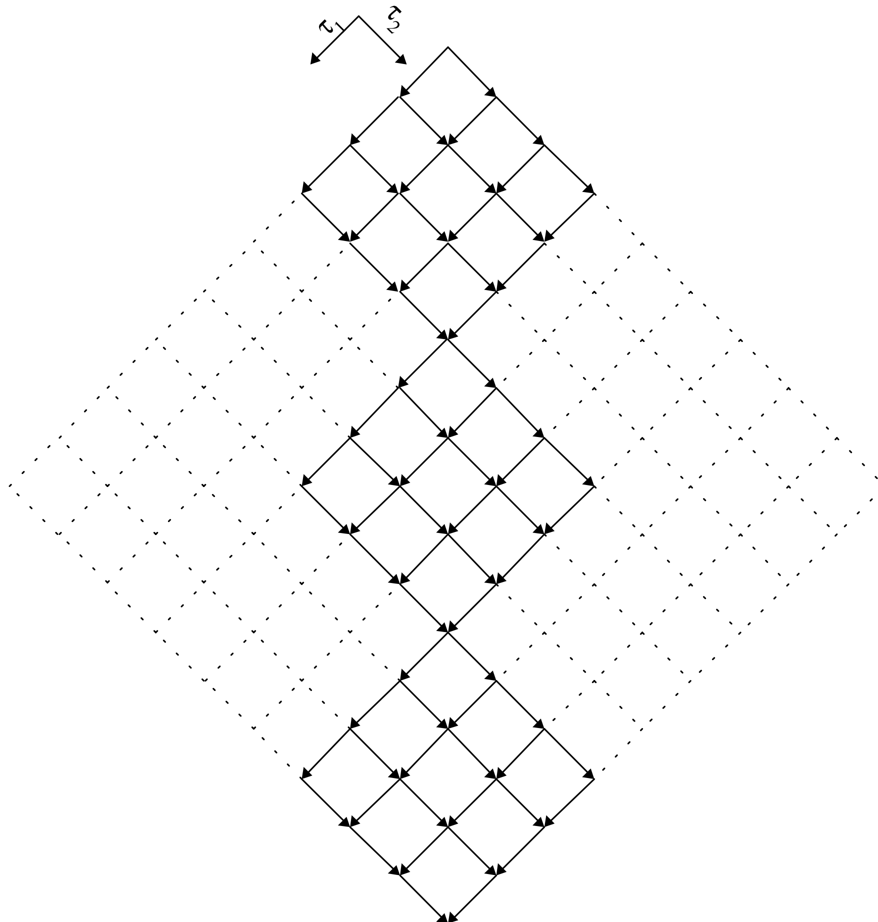


figure IV-2-5 : le GM de S_2 n'occupe qu'un tiers de l'hyperpavé le bornant

La partie de l'hyperpavé occupée par le *GM* est d'autant plus petite relativement à la borne théorique que le nombre de requêtes augmente, car chaque tâche doit respecter les contraintes temporelles qui lui sont affectées. De plus, le *GM* diminue fortement lorsque le délai critique des tâches est court (i.e. tâches non à échéance sur requête). Les contraintes de précédence ainsi que les exclusions mutuelles diminuent encore le *GM*. Cependant, les diminutions induites par les contraintes temporelles et structurelles ne suffisent pas à garantir que le *GM* est constructible.

De plus, même si les contraintes temporelles diminuent la taille finale du *GM*, sa construction entraîne des retours arrières car, étant donné un préfixe valide d'ordonnancement, il n'est pas dit qu'il puisse mener à un ordonnancement valide. Un autre de nos centres d'intérêt est de limiter le nombre de retours arrières lors de la construction du *GM*.

IV-2.3.2 Structure de données utilisée pour stocker et rechercher les marquages

Le graphe ne contenant qu'une seule instance d'un marquage donné, il faut choisir une structure de données facilitant la recherche et le stockage de marquages. Cette structure peut être un arbre binaire.

> Définition IV-2-2 : Un **arbre binaire** ou ***AB*** de hauteur *h* est un arbre dont chaque nœud *N* a au plus 2 fils. Par convention, le premier, nommé fils gauche, est relié au nœud *N* par un arc étiqueté par *0*, le second, nommé fils droit, lui est relié par un arc étiqueté par 1.

Un ensemble $\{i_1,i_2,\ldots,i_k\}$ sans répétition d'entiers positifs avec $0 \le i_{j,j=1..k} \le max-1$ peut être représenté aisément par un *AB* : chacun des $i_{j,j=1..k}$ est codé en binaire, sur $\lceil log_2(max) \rceil$. Un entier est présent dans l'*AB* si et seulement si la séquence binaire correspondante y est aussi. Le test d'existence d'un entier dans l'*AB* s'effectue donc en un temps $\theta(\lceil log_2(max) \rceil)$, et l'insertion d'un entier nécessite le même temps.

Par exemple, la figure IV-2-6 représente un arbre binaire qui implémente un ensemble {0,3,5,6,7} inclus dans l'ensemble {0,1,2,3,4,5,6,7}. Chaque entier est implicitement codé par $\lceil log_2(8) \rceil = 3$ bits : un arc vers la gauche signifie *0*, alors qu'un arc vers la droite signifie 1. 3 étant codé en binaire sur 3 bits par 011, le chemin correspondant dans l'*AB* est gauche, droite, droite.

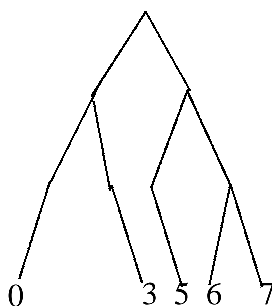


figure IV-2-6 : un arbre binaire pouvant représenter les entiers de 0 à 7

La structure d'*AB* est peu coûteuse en temps pour l'ajout, la suppression, et le test d'existence d'une valeur dans l'ensemble implémenté. Il en est de même pour l'espace de

stockage nécessaire. C'est donc cette structure que nous choisissons pour stocker l'ensemble des marquages du RdP.

Le marquage d'une place p_i colorée est constitué d'un $|C|$-uplet $(m_{i,c_1}, m_{i,c_2},\ldots, m_{i,c_{|C|}})$, avec C l'ensemble des couleurs. Les places étant bornées par la structure du réseau, nous appelons $capacité(i,c_j)$ le nombre maximal de jetons de couleur c_j dans la place p_i. Le marquage de p_i peut donc être représenté par une branche de longueur $\sum_{c\in C}\lceil log_2(capacité(i,c)+1)\rceil$ bits dans un AB : cela correspond à mettre bout à bout le nombre de jetons de chaque couleur contenue dans p_i, chaque couleur étant représentée par $\lceil log_2(capacité(i,c)+1)\rceil$ bits.

Un marquage du modèle RdP peut donc être représenté dans un AB par $h=\sum_{p_i\in Q}\sum_{c\in C}\lceil log_2(capacité(i,c)+1)\rceil$, ce qui ne peut pas être supérieur à $\sum_{p_i\in Q}\sum_{c\in C}log_2(2\times(capacité(i,c)+1))=log_2(\prod_{p_i\in Q}\prod_{c\in C}2\times capacité(i,c)+2)$ bits.

Tout chemin d'un AB représentant un marquage a une longueur h, l'AB associé a donc une hauteur $h_{AB}=h+1$, et l'insertion et la suppression d'un marquage coûtent h_{AB}.

La taille maximale nécessaire au stockage de l'ensemble des marquages accessibles sera donc en $\theta(\sum_{i=0}^{h_{AB}-1} 2^i)$ puisque chaque nœud de hauteur inférieure à h_{AB} a exactement 2 fils, ce qui se simplifie en $\theta(2^{h_{AB}}-1)$.

Chaque feuille de l'AB correspond donc à un marquage accessible du RdP. Parallèlement, on construit le graphe contenu dans un hyperpavé, et une bijection relie chaque feuille (marquage) de l'AB à chaque nœud de l'hyperpavé qui, rappelons-le, ne peut avoir plus de $(1+n_c)(r+2P)^{n+1}\prod_{i=0}^{n}\frac{C_i}{P_i}$ nœuds. Chaque nœud de l'hyperpavé a une taille constante, l'espace requis par un GM est donc au pire en $\theta(2^{h_{AB}}-1+(1+n_c)(r+2P)^{n+1}\prod_{i=0}^{n}\frac{C_i}{P_i})$, ce qui vaut $\theta(\prod_{p_i\in Q,c\in C}(2\times capacité(i,c)+2)+ (1+n_c)(r+2P)^{n+1}\prod_{i=0}^{n}\frac{C_i}{P_i})$.

L'obtention de h_{AB} est relativement aisée puisque chaque place du modèle est bornée, soit par la structure du réseau (cas des $P_{i,j,k}$ qui sont des places sauves, sauf les $P_{i,j,2}$ qui peuvent contenir jusqu'à $b_{i,j}-2$ jetons), soit par le fonctionnement avec la règle de tir maximal (cas des places $time_i$ bornées par P_i), soit par les contraintes de marquages (les places $activ_i$ ne peuvent contenir plus d'un seul jeton a). Les places ressources, ainsi que la place processeur, sont bornées par leur marquage initial, idem pour les places $wait_i$.

Pour les places boîtes aux lettres, on ne peut démontrer la bornitude à partir du réseau mais à partir du système modélisé : étant donné qu'il y a isomorphisme entre le modèle et le système de tâches, et a fortiori entre les boîtes aux lettres du système et les places boîtes aux lettres, le raisonnement s'opère sur les messages entre tâches. Etant donné que les périodes d'émission et de réception des messages sont identiques, les places boîtes aux lettres sont bornées par l'écart maximal qui peut exister entre le nombre d'émissions et de réceptions. Cet écart donne une borne sur le nombre de jetons que peut contenir une boîte aux lettres. D'où un algorithme de calcul d'une borne sur le nombre de jetons contenu dans une boîte aux lettres. La *date_au_plus_tôt*(*instr*) est donnée par le temps nécessaire à l'exécution des blocs précédant l'instruction *instr* dans la tâche concernée. La *date_au_plus_tard*(*instr*) est donnée par le délai critique de la tâche concernée moins le temps nécessaire à l'exécution des blocs suivant *instr*.

Plusieurs tâches sont susceptibles de prendre et/ou déposer des messages dans une ou plusieurs boîtes aux lettres. De plus, chaque tâche n'est pas limitée à un seul accès à la boîte aux lettres par occurrence. Afin de faciliter l'étude de la bornitude d'une place boîte aux lettres, nous abstrayons les tâches communicantes en sous-tâches qui sont soit émettrices, soit récep-

trices. Soit b une boîte aux lettres. Chaque instruction *envoi*=`déposer_BAL(b)` d'une tâche τ_i donne naissance à un couple (s_{e_j},p_{e_j}) avec $s_{e_j}=r_i+date_au_plus_tôt(envoi)$ et $p_{e_j}=P_i$. Les couples sont donc choisis tels qu'ils correspondent à des émissions au plus tôt (productions de messages dans la place correspondante) des messages vers b. Dualement, chaque instruction *réception*=`retirer_BAL(b)` de τ_i donne naissance à un couple (s_{r_j},p_{r_j}) avec $s_{r_j}=r_i+date_au_plus_tard(réception)$ et $p_{r_j}=P_i$. Les couples sont donc choisis tels qu'ils correspondent à des réceptions (consommations de jetons dans la place correspondante) au plus tard des messages de b.

Théorème IV-2-2 : Soit b une boîte aux lettres et soient $(s_{e_1}, p_{e_1}),\ldots,(s_{e_k}, p_{e_k})$ les couples d'émission au plus tôt tels que un message peut être émis vers b au plus tôt à la date s_{e_i}, puis ré-émis périodiquement toutes les p_{e_i} unités de temps. Dualement, soient $(s_{r_1}, p_{r_1}),\ldots,(s_{r_k}, p_{r_m})$ les couples de réceptions de messages au plus tard de la boîte aux lettres. Le nombre de messages présents dans b ne peut en aucun cas excéder :

$$max_{x\in\{0..max\{s_i,s_j\}+ppcm(p_i,p_j)\},i=\{e_1,\ldots,e_k\},j=\{r_1,\ldots,r_m\}}$$

$$\left\{ \sum_{i\in\{e_1,\ldots,e_k\}} \max\left\{0,\left\lceil \frac{x-s_i}{p_i}\right\rceil\right\} - \sum_{j\in\{r_1,\ldots,r_m\}} \max\left\{0,\left\lfloor \frac{x-s_j}{p_j}\right\rfloor\right\} \right\}.$$

Preuve :

Soit une date x, le nombre de messages présents dans une boîte aux lettres ne peut excéder $borne(b,x)=\sum_{i\in\{e_1,\ldots,e_k\}} \max\left\{0,\left\lceil \frac{x-s_i}{p_i}\right\rceil\right\} - \sum_{j\in\{r_1,\ldots,r_m\}} \max\left\{0,\left\lfloor \frac{x-s_j}{p_j}\right\rfloor\right\}$. En effet, puisque les couples (s_{e_i}, p_{e_i}) représentent les émissions au plus tôt et que les couples (s_{r_j}, p_{r_j}) représentent les réceptions au plus tard, $borne(b,x)$ représente l'écart maximal pouvant exister entre le nombre de messages émis et le nombre de messages consommés dans la boîte aux lettres b.

Le fait que l'on n'étudie cette borne que sur une période de 0 au *ppcm* des périodes des tâches communicantes se servant de b est dû au fait que les fréquences d'émission et de réception de messages vers une boîte aux lettres sont identiques. Cela implique que $\sum_{i=\{e_1,\ldots,e_k\}} 1/p_i=\sum_{j=\{r_1,\ldots,r_m\}} 1/p_j$, d'où pour $x\geq max_{i=\{e_1,\ldots,e_k\},j=\{r_1,\ldots,r_m\}}\{s_i,s_j\}$, en utilisant la définition de $borne(b,x)$, et le fait que pour tout couple d'entiers (y,z), $ppcm(y,z)/y$ est un entier :

$borne(b,x+ppcm_{i=\{e_1,\ldots,e_k\},j=\{r_1,\ldots,r_m\}}(p_i,p_j))=borne(b,x)$

❑

Cet algorithme n'est pas polynomial puisqu'il nécessite l'étude des messages sur un *ppcm* des périodes des tâches communicantes. Cependant, ce n'est pas significatif par rapport à la méthodologie d'obtention des séquences qui se base sur la construction d'un graphe de hauteur de l'ordre du *ppcm* des périodes.

IV-2.4 Obtention des séquences d'ordonnancement valides

Rappelons que le langage du modèle RdP est exactement l'ensemble de toutes les séquences d'ordonnancement valides du système de tâches modélisé. Il nous faut donc extraire ce langage : cela est fait par construction du graphe des marquages.

IV-2.4.1 Construction du graphe des marquages

L'algorithme, donné en annexe I-3, se base sur une simulation du RdP avec la règle de tir maximal, et une construction du graphe soit en profondeur d'abord, soit en largeur. Si tout ou partie du *GM* doit être construite, alors c'est une construction en largeur que nous utiliserons, alors que si on est seulement intéressé par une séquence valide quelles que soient ses caractéristiques, une construction en profondeur, nécessitant moins d'espace qu'une construction en largeur, sera choisie.

Le principe de la construction est simplifié par rapport aux réseaux de Petri classiques :

- Chaque place est bornée, ce qui permet une représentation efficace de l'ensemble des marquages accessibles par arbre binaire.
- On sait d'avance que la profondeur du *GM* est bornée par t_c+P+1 avec t_c date du dernier temps creux acyclique et P méta-période du système de tâches. La date t_c est calculée avant la construction du *GM*, puisqu'elle sert aussi à déterminer la date de réveil de la tâche oisive et la nombre de temps creux acycliques présents dans les séquences d'ordonnancement. Remarquons qu'un modèle RdP simplifié, que nous notons RdP_2, permet d'obtenir t_c de la façon suivante : RdP_2 est construit parallèlement au modèle RdP correspondant au système de tâches à ordonnancer. RdP_2 intègre la périodicité des tâches, ainsi que les communications mais pas les contraintes de ressources ni de non préemption puisque celles-ci n'entrent pas en compte dans le calcul de t_c. Les contraintes temporelles ne sont pas non plus prises en compte. Un mot de longueur $r+P$ sur RdP_2 (n'intégrant pas encore la tâche oisive) est alors construit : il correspond à un ordonnancement conservatif, avec respect des contraintes de précédence engendrées par les communications. Les tirs étiquetés par le mot vide correspondent alors aux temps creux donnés par tout algorithme conservatif. Les dates d'occurrence des temps creux sont mémorisées dans une table ; il ne reste plus alors qu'à différencier les temps creux acycliques des temps creux cycliques à l'aide de la méthode décrite en section III-2. La date t_c nous sert à déterminer r_0=t_c+1, et le nombre de temps creux acyclique donne le marquage initial de la place P_c du modèle RdP.

IV-2.4.1.a Construction en largeur

Le premier nœud du graphe est le marquage initial M_0 à la hauteur *0*. Soient E_1, E_2,…,E_k les ensembles maximaux de transitions franchissables (en suivant le mode de fonctionnement maximal du RdP), k nœuds sont ajoutés à la hauteur 1 et sont connectés à M_0. Dans le cas monoprocesseur, la ressource processeur impose $k \leq n$.

Le principe de construction du graphe en largeur est illustré sur la figure IV-2-7.

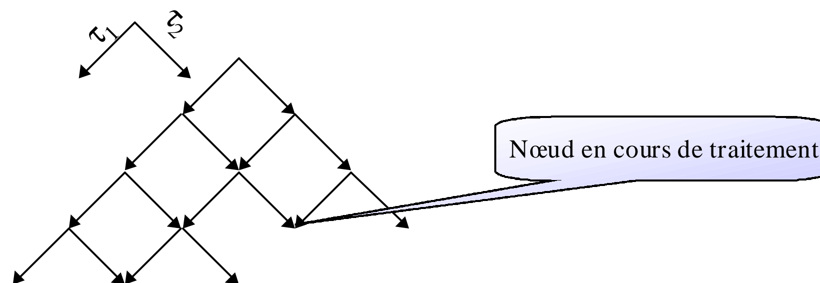


figure IV-2-7 : construction du graphe en largeur pour un système de 2 tâches

Pour chacun des nœuds situés à la hauteur *h*, on construit de la même façon chacun des nœuds fils qui seront placés à la hauteur $h+1$, un nœud n'étant bien entendu créé que si il est valide (i.e. il fait partie de l'ensemble des marquages terminaux) et si il n'existe pas déjà dans la graphe. La vérification de l'existence ou non d'un marquage dans un ensemble de marquages peut se faire en un temps h_{AB} (voir section IV-2.3.2) qui est un logarithme du produit des bornes des places du RdP. Le graphe est ainsi construit hauteur après hauteur jusqu'à la profondeur de simulation t_c+P+1. Il n'y a qu'un marquage à cette date, de plus, ce marquage est le premier marquage qui était déjà présent dans le graphe à une hauteur différente, en l'occurrence à la date t_c+1.

IV-2.4.1.b Construction en profondeur

Le premier nœud du graphe est toujours le marquage initial M_0 à la hauteur *0*. Soient E_1, $E_2,\ldots,E_k$ les ensembles maximaux de transitions franchissables, le traitement de M_0 entraîne la construction d'un nœud $M_{1,1}$ à la hauteur 1, avec $M_0(E_1\rangle M_{1,1}$. Les fils de $M_{1,1}$ sont alors traités à leur tour récursivement jusqu'à la profondeur t_c+P+1, avant que le nœud $M_{1,2}$ (avec $M_0(E_2\rangle M_{1,2}$) ne soit construit à la hauteur 1 et traité à son tour.

Le principe de construction du graphe en profondeur est illustré sur la figure IV-2-8.

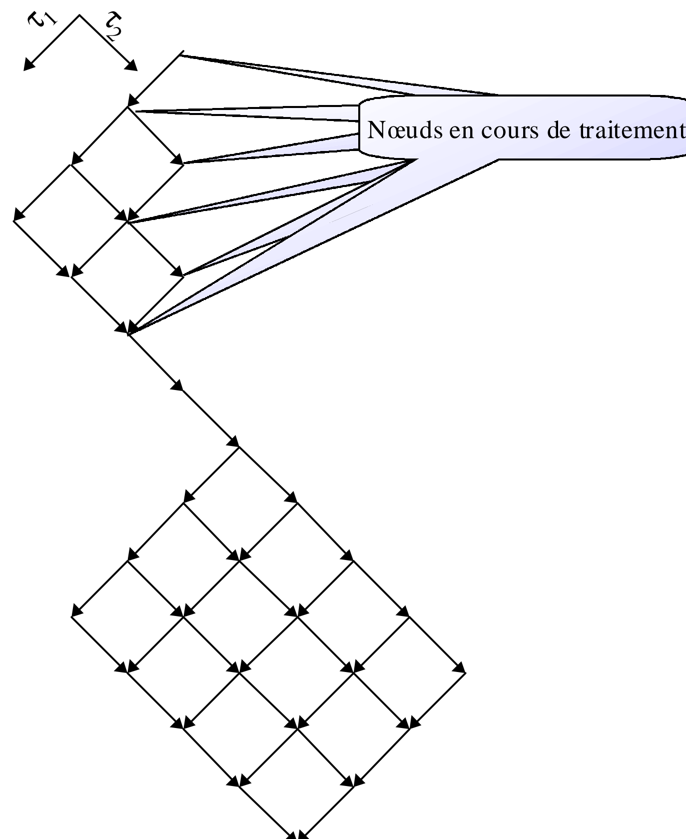


figure IV-2-8 : construction du graphe en profondeur pour un système de 2 tâches

IV-2.4.1.c Comparatif d'une construction en largeur par rapport à en profondeur

Dans les deux cas, tout marquage non valide, c'est à dire ne faisant pas partie des marquages terminaux, est écarté. Un marquage ne menant à aucun marquage terminal (nous parlons

de marquage stérile) doit aussi être écarté puisqu'il ne fait pas partie du centre du langage. Ce mode d'obtention du langage entraîne des retours arrières (*backtracking*), puisqu'un marquage inséré dans le graphe à un instant donné peut s'avérer stérile plus tard.

Les avantages d'une construction en largeur par rapport à une construction en profondeur d'un graphe des marquages sont les suivants :

- Etant donné que la profondeur du graphe des marquages est connue, l'état d'avancement de la construction du graphe est connu par rapport à la profondeur attendue, ainsi l'indice de progression de la construction peut être présenté à l'utilisateur de l'outil. Le calcul pouvant être long, l'utilisateur est alors à même d'arrêter la construction d'un graphe qui s'avèrerait trop volumineux afin d'imposer des contraintes supplémentaires au système de tâches pour diminuer la taille du graphe et ainsi accélérer le processus de construction.
- Il n'est pas nécessaire de conserver jusqu'à la fin de la construction les nœuds non valides. En effet, l'inconvénient majeur d'une construction en profondeur d'abord est le suivant : soit un nœud N stérile, de hauteur h, dont le traitement en profondeur entraîne la construction d'un nombre de nœuds conséquents avant que l'on ne s'aperçoive du fait que tous ses nœuds fils conduisent à des fautes temporelles. Si N est supprimé physiquement du graphe, c'est à dire oublié, lorsque l'on s'aperçoit de sa non appartenance au langage terminal, N peut être étudié entièrement à nouveau pour chacun de ses autres nœuds père. Pour palier ce problème, l'une des solutions est de laisser dans le graphe les nœuds non valides, le problème étant bien sûr la place occupée par ces nœuds. Ce problème ne se pose pas en largeur, puisque tout nœud n'est étudié qu'une fois, et non pas n fois s'il a n nœuds pères.
- L'inconvénient majeur d'une construction en largeur est la nécessité de l'obtention exhaustive du graphe avant d'obtenir un mot du langage. En effet, avant d'arriver à la profondeur t_c+P+1, les préfixes de mots présents dans le graphe des marquages ne sont pas forcément des préfixes de mots du centre du langage terminal.

Le choix de construire le graphe en largeur ou en profondeur est donc laissé à l'initiative de l'utilisateur :

- S'il veut une séquence d'ordonnancement quelconque (ou presque), une construction en profondeur s'impose, avec ou sans conservation des marquages non valides suivant le choix de l'utilisateur.
- Si au contraire, il veut pouvoir choisir des séquences sur des critères précis, tout ou partie du graphe doit être construite, impliquant le choix d'une construction en largeur.

IV-2.4.2 Diminution des cas de retours arrière

Nous avons vu dans la section IV-2.4.1.c que les cas de retours arrière existaient, surtout dans le cas de la construction en profondeur sans conservation des nœuds non valides. Il nous faut donc limiter au maximum ces cas en détectant au plus tôt la non validité des nœuds. La méthode que nous employons consiste à détecter à l'avance qu'un marquage est stérile (i.e. il ne peut pas conduire à une séquence valide).

On distingue deux cas de retour arrière : l'interblocage, relativement difficile à détecter, et le non respect des contraintes (un interblocage mènera de toutes façons au non respect des contraintes). La détection du non respect des contraintes peut s'effectuer par la vérification pour tout marquage du respect de conditions suffisantes d'ordonnançabilité basées sur les paramètres temporels dynamiques des tâches du système.

Rappelons que la latence dynamique à l'instant t de la $k^{ème}$ instance d'une tâche τ_i de prochaine échéance $d_{i,k}$, est $L_i(t)=d_{i,k}-t$, en d'autres termes, le temps restant à τ_i avant sa prochaine échéance.

Pour une configuration de n+1 occurrences courantes de tâches indépendantes dont on connaît les dates de réveil, ordonnées de telle façon que l'échéance de τ_i tombe avant l'échéance de τ_j si $i<j$ (i.e. elles sont numérotées selon leur échéance), on peut dégager une CNS d'ordonnançabilité pour l'algorithme *earliest deadline* (algorithme optimal dans ce cas) qui est donnée par $\forall k \in 0..n$, $\sum_{i=0}^{k} C_i(t) \leq L_k(t)$ **(1)** avec $C_i(t)$ durée de τ_i à traiter à l'instant t. En effet, lorsque l'échéance de τ_k tombe, les échéances des tâches $\tau_{i,i=0..k-1}$ sont déjà tombées, et toutes les tâches *0* à k doivent donc avoir été traitées.

A partir d'un état donné, une condition nécessaire d'ordonnançabilité des prochaines occurrences des tâches est **(1)**. L'avantage de cette condition est qu'elle peut être vérifiée, à partir d'une table ordonnée suivant les latences croissantes (voir tableau IV-2-1), en $\theta(n)$, et que lors d'une construction du graphe des marquages, l'obtention de la table triée sur la latence s'effectue en $\theta(n)$ à partir de la table du nœud père.

Soit S un système de n+1 tâches numérotées de *0* à n. La table ordonnée de laxité dynamique à l'instant t est donnée sur le tableau IV-2-1.

$\tau_{t,0}$	$\tau_{t,1}$	…	$\tau_{t,n}$
$C_{t,0}(t)$	$C_{t,1}(t)$	…	$C_{t,n}(t)$
$L_{t,0}(t)$	$L_{t,1}(t)$	…	$L_{t,n}(t)$

tableau IV-2-1 : table des latences croissantes

La condition nécessaire **(1)** est vérifiée pour chaque marquage de hauteur t (en numérotant les hauteurs du graphe à partir de 0) par l'algorithme très simple et rapide :

```
ChargeTotale ← 0
Pour i de 0 à n faire
      ChargeTotale ← ChargeTotale + Ct,i(t)
      Si ChargeTotale > Lt,i(t) alors
            Renvoyer Configuration_non_ordonnançable
      FinSi
Fait
Renvoyer Configuration_peut_être_ordonnançable
```

Entre deux marquages, la tâche τ_j se voit attribuer le processeur, la table est alors mise à jour de la façon suivante :

– Toutes les latences diminuent de 1.
– Toutes les charges dynamiques sont identiques, sauf en ce qui concerne τ_j.
– Si τ_j est terminée, alors la prochaine occurrence de τ_j est réinsérée dans la table à sa place, ce qui nécessite un temps $\theta(n)$.

L'ordre des tâches reste inchangé, sauf en ce qui concerne τ_j dans le cas où celle-ci vient de se terminer, ce qui permet de conclure que la complexité de la vérification de cette contrainte et de la mise à jour de la table est en $\theta(n)$.

Ce principe pourrait d'ailleurs être repris pour coupler, sans augmenter la surcharge processeur, l'algorithme *earliest deadline* à une heuristique de choix de tâches à éliminer lors des surcharges afin d'éviter le non respect en dominos des échéances.

Les gains en temps et en espace apportés par cette heuristique sont difficilement quantifiables, cependant, on peut toutefois estimer leur apport sur des exemples simples (voir figure IV-2-9).

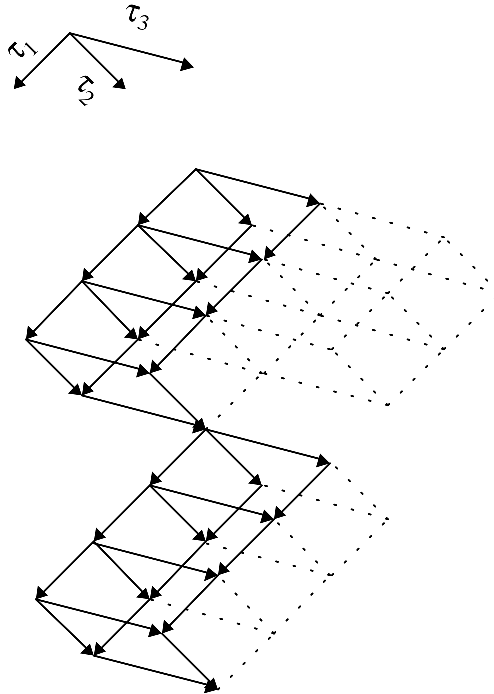


figure IV-2-9 : gain apporté par l'utilisation de la table de latences lors de la construction d'un graphe des marquages pour le système de tâches indépendantes S={ τ_1<0,3,5,5>, τ_2<0,1,5,5>, τ_3<0,2,10,10>} : le graphe des marquages est représenté par les flèches, les pointillés représentent ce qui est construit en sus en l'absence d'optimisation par la table de latence

Sur cet exemple, l'utilisation de la table de latence permet d'obtenir directement le langage terminal du RdP, sans retour arrière, alors que les retours arrière qui auraient lieu sans cette table sont représentés en pointillés.

IV-2.4.3 Diminution de la taille du graphe des marquages

IV-2.4.3.a Contraintes de successeur

Les contraintes de successeur limitent les entrelacements inutiles entre les tâches : deux occurrences de tâches indépendantes τ_i et τ_j peuvent en effet se préempter mutuellement jusqu'à C_i+C_j-1 fois.

Pour limiter la taille du graphe des marquages, en diminuant les entrelacements entre les tâches qui se préemptent, on interdit à toute tâche d'interrompre la tâche travaillant sauf dans les cas suivants :

1. Une tâche qui vient d'être (ré)activée peut préempter la tâche travaillant
2. Une tâche qui vient d'être débloquée par la réception d'un message ou la libération d'une ressource peut préempter la tâche travaillant
3. Si la tâche travaillant demande son entrée en section critique, toute tâche peut la préempter.

Ces contraintes permettent d'éviter tout entrelacement inutile de parties de tâches indépendantes. En effet, si deux parties indépendantes de tâches sont indépendantes, il est inutile de les faire s'entrelacer, et on peut se contenter de les exécuter séquentiellement sans nuire à l'ordonnançabilité du système.

L'ensemble des ordonnancements d'un système de tâches {τ_1<0,5,11,11>, τ_2<0,6,11,11>} représenté sur la figure IV-2-10 est ainsi fortement réduit : les pointillés représentent le graphe sans les contraintes, alors que seule la partie représentée en gras est retenue dans un graphe construit avec les contraintes de successeur.

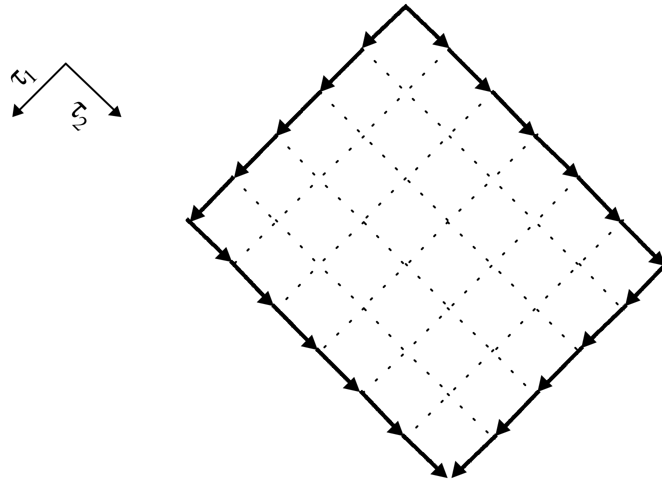


figure IV-2-10 : gain apporté par l'utilisation des contraintes de successeur sur un système de 2 tâches indépendantes simultanées

IV-2.4.3.b Contraintes absolues

Les contraintes absolues permettent une diminution du graphe des marquages a priori. Ce sont des contraintes basées sur le temps de réponse des tâches. Le concepteur peut en effet vouloir imposer que le temps de réponse appartienne à un intervalle donné, afin par exemple de forcer une diminution de la gigue temporelle ou tout simplement de limiter l'espace initial de recherche de séquences d'ordonnancement.

Chaque contrainte absolue est exprimée sous la forme d'une contrainte mathématique, dont les opérandes sont :

– soit des constantes physiques d'une tâche (date de réveil, charge, délai critique, période),
– soit fonction d'un nœud du graphe, à partir duquel on cherche à évaluer le prochain temps de réponse d'une tâche, ou des bornes sur cette date.

Les fonctions dépendantes d'un nœud du graphe sur lequel on les évalue sont les suivantes :

– *TR* : *nom*×*nœud*→*entier*, fournit une borne sur le prochain temps de réponse de l'instance courante de la tâche τ_{nom} à partir d'un *nœud* considéré du *GM*. Ce *nœud* correspond à l'état du système après un préfixe d'ordonnancement (chemin du nœud initial du *GM* au *nœud*), et il est intéressant de trouver à partir de cet état quel peut être le temps de réponse minimal de l'instance courante de τ_{nom}.
– *L* : *nom*×*nœud*→*entier*, fournit la laxité (ou temps de retard admissible lorsqu'on se place après la terminaison de la tâche) : $L(\tau_{nom},N)=(D_{nom}-TR(\tau_{nom},N))$.
– *TxR* : *nom*×*nœud*→*entier*, fournit le taux de réaction de τ_{nom} : $TxR(\tau_{nom},N)=TR(\tau_{nom},N)/D_{nom}$.

---

exemple IV-2-3 : une contrainte peut être donnée sous la forme :

$\forall N\in G$, $TR(\tau_i, N)\leq 1+C_i$, avec G l'ensemble des nœuds du graphe. Cette contrainte impose que le temps de réponse de chaque occurrence de la tâche τ_i ne soit pas plus grand que 1 plus sa charge.

La grammaire de description des contraintes en syntaxe BNF, où les mots clés sont en caractères gras et les commentaires sont donnés entre /* et */, est donnée ci-dessous.

```
contrainte ::= ( contrainte ) | contrainte & contrainte | équation
/* L'opérateur & est l'opérateur logique « et » */
équation ::= entier comparateur entier | rationnel comparateur rationnel
/* Les équations sont soit entières, soit rationnelles */
entier ::= fonction_entière | nombre_entier | (floor) rationnel
          | (ceil) rationnel | entier opérateur entier | ( entier )
/* floor (resp. ceil) est la partie entière supérieur (resp. inférieure)*/
rationnel ::= fonction_rationnelle | nombre_rationnel | (rat) entier
          | rationnel opérateur rationnel | ( rationnel )
/* rat est la fonction de coercion entier → rationnel */
comparateur ::= > | >= | < | <= | == | !=
opérateur ::= + | - | * | /
fonction_entière ::= ∀N.TR ( nom_de_tâche,N ) | ∀N.L ( nom_de_tâche,N )
             | r_nom_de_tâche | C_nom_de_tâche
             | D_nom_de_tâche | P_nom_de_tâche
fonction_rationnelle ::= ∀N.TxR (nom_de_tache,N )
nombre_entier ::= [0-9]+
nombre_rationnel ::= [0-9]+.[0-9]+
```

Les contraintes sont évaluées pour chaque nœud construit. Si l'une d'entre elles n'est pas respectée, alors le nœud est déclaré non valide et abandonné.

Si l'ensemble des contraintes ne contient aucune expression dépendante des nœuds, alors le graphe est soit vide (cas où la contrainte est toujours fausse), soit identique à ce qu'il serait sans contrainte (cas où la contrainte est une tautologie). Dans ce cas, il est évident que les contraintes ne servent à rien.

Comme précédemment, pour limiter les retours en arrière, il est préférable de déterminer au plus tôt la non validité d'un nœud. A l'aide de la table décrite en §IV-2.4.2, il est possible, à tout moment (i.e. à chaque nœud du graphe des marquages), de déterminer un intervalle auquel appartiendra le temps de réponse de l'occurrence d'une tâche correspondant au nœud considéré. Sa borne supérieure est donnée par son délai critique, puisqu'on peut a priori retarder une tâche aussi loin que l'on veut dans le cadre du respect de son échéance. Sa borne inférieure est calculée de la façon suivante à partir de la table des latences croissantes, les notations utilisées étant en partie tirée du tableau IV-2-1 :

```
Fonction TempsDeRéponseMinimal(N:Noeud, τi:nom_de_tâche) Renvoie entier
      Min,t : entier
      τj,indice : nom_de_tâche
Début
      t←hauteur(N)
      Min←Cτi(t)
      indice←première tâche de la table des latences croissantes
      Tant que Min+Cindice(t)>Lindice(t) Faire
```

```
            Min←Min+Cindice(t)
            indice← tâche suivante dans la table
      Fait
      Renvoyer Min+t-max(0, ⌊(t-ri)/Pi⌋ Pi)
Fin
```

La boucle se termine obligatoirement car la table des latences satisfait la condition nécessaire d'ordonnançabilité donnée en IV-2.4.2. L'algorithme se base sur les propriétés suivantes :

Au début, `Min` reçoit la charge restant à traiter pour τ_i, cela correspond au cas où τ_i est exécutée à partir du nœud `N` sans être préemptée jusqu'à sa terminaison. Puis on parcourt la table dans l'ordre croissant des latences. Considérons $\tau_{t,0}$ la tâche de latence minimale au temps *t*. Si c'est la tâche τ_i, alors puisque la table respecte la condition nécessaire d'ordonnançabilité, la tâche τ_i peut être, hors réveil d'autres tâches, exécutée immédiatement, et son temps de réponse minimal est donc sa charge restant à traiter plus le temps déjà écoulé depuis son réveil. Si $\tau_{t,0}$ n'est pas τ_i, alors si les deux tâches peuvent être exécutées avant l'échéance de $\tau_{t,0}$, τ_i peut être exécutée immédiatement, sinon elle devra attendre la terminaison de $\tau_{t,0}$. Le même raisonnement s'applique pour toutes les tâches de la table jusqu'à ce que les latences soit suffisantes pour permettre d'exécuter τ_i, ce qui sera le cas au pire à la position de la tâche τ_i.

La complexité du calcul de l'intervalle possible du temps de réponse d'une tâche en un nœud *N* est linéaire en fonction du nombre de tâches, ce qui donne la complexité du calcul des fonctions dépendantes du nœud.

Ces contraintes permettent de limiter la taille du graphe des marquages, tout en coupant les mauvaises branches le plus tôt possible.

Il reste maintenant à exprimer des contraintes qualitatives. Celles-ci sont appliquées après la construction du graphe des marquages du réseau de Petri.

IV-2.4.4 Extraction de séquences d'ordonnancement optimales

Le graphe des marquages construit, il reste à choisir une séquence valide, en se basant sur certains critères qualitatifs. Un critère qualitatif est relatif à l'ensemble des séquences d'ordonnancement valides. Notre but est de trouver l'ensemble des séquences d'ordonnancement valides optimisant les critères choisis par le concepteur d'une application temps réel.

Chaque critère qualitatif s'applique sur un ensemble de tâches $E_{opt}=\{\tau_{i_1}, \tau_{i_2},\ldots, \tau_{i_k}\}\subseteq S$ sur lesquelles on veut l'optimiser, au pire ou en moyenne. Afin de simplifier les notations, chaque tâche τ_i de E_{opt} est décomposée en ses $\left\lfloor \frac{t_c+P+1}{P_i} \right\rfloor$ instances respectives, formant un nouvel ensemble noté $E'_{opt}=\{\tau_{j_1}, \tau_{j_2},\ldots, \tau_{j_m}\}$ avec $m\geq k$, d'échéances respectives $\{d_{j_1}, d_{j_2},\ldots, d_{j_m}\}$, de temps de réponse respectifs $\{tr_{j_1}, tr_{j_2},\ldots, tr_{j_m}\}$. Les critères retenus sont les suivants :

- Minimisation du temps de réponse.
  - Au pire : trouver l'ensemble des séquences minimisant $max_{\tau\in E'_{opt}}\{tr_\tau\}$
  - En moyenne : trouver l'ensemble des séquences minimisant $moyenne_{\tau\in E'_{opt}}\{tr_\tau\}$
- Maximisation de la latence.
  - Au pire : trouver l'ensemble des séquences maximisant $min_{\tau\in E'_{opt}}\{d_\tau-tr_\tau\}$
  - En moyenne : trouver l'ensemble des séquences maximisant $moyenne_{\tau\in E'_{opt}}\{d_\tau-tr_\tau\}$

- Minimisation du taux de réaction.
  - Au pire : trouver l'ensemble des séquences minimisant $max_{\tau\in E'_{opt}}\{tr_\tau/D_\tau\}$
  - En moyenne : trouver l'ensemble des séquences minimisant $moyenne_{\tau\in E'_{opt}}\{tr_\tau/D_\tau\}$
- Exécution au plus tôt : on cherche à exécuter au plus tôt, parmi les séquences valides, les tâches de E'_{opt}. Pour formaliser, notons $E''_{opt}=\{\tau_{j_{1,1}},\ldots, \tau_{j_{1,C_1}}, \ldots, \tau_{j_{m,1}}, \tau_{j_{m,C_m}}\}$ qui est l'ensemble constitué des tâches de durée unitaire composant E'_{opt}. De la même manière qu'aux tâches de E'_{opt}, on leur affecte un ensemble de temps de réponse tr_τ. Il s'agit alors de minimiser $\Sigma_{\tau\in E''_{opt}}\{tr_\tau\}$.

Il est important de constater que les critères en moyenne portent sur la moyenne des critères sur les instances des tâches et non pas sur les tâches elles-mêmes. De plus, si l'ensemble de départ des séquences d'ordonnancement considérées est non vide, alors l'ensemble des séquences après application d'un critère à optimiser est un sous-ensemble non vide de l'ensemble de départ.

On peut noter que :

- Pour un nœud donné du graphe des marquages, ces propriétés sont indépendantes des chemins entrant et sortant, dans le sens que les critères sont évalués de la même façon pour un nœud quel que soit le chemin entrant ou sortant considéré.
- Basés sur le temps de réponse des tâches, les critères permettent au concepteur d'obtenir une séquence d'ordonnancement dont les tâches sélectionnées auront toujours un temps de réponse court.

L'obtention des séquences optimales au vu des critères choisis est linéaire en fonction de la taille (nombre de nœuds plus nombre d'arcs) du graphe des marquages. Elle se base sur une recherche de plus court chemin par parcours topologique dans un graphe sans cycle avec une source et une destination : le graphe des marquages. C'est un cas particulier de l'algorithme de Ford [For 56][GM 95] appliqué à un graphe sans cycle topologiquement trié. Il suffit de pondérer les arcs du *GM* de façon à ce qu'un chemin de coût minimum soit optimal.

IV-2.4.4.a Exécution au plus tôt

Afin de trouver les séquences dans lesquelles un ensemble de tâches $E_{opt}=\{\tau_{i_1}, \tau_{i_2},\ldots, \tau_{i_k}\}$ est exécuté au plus tôt, le coût d'un arc du *GM* est choisi tel que :

$$Coût_AuPlusTôt(Arc(N_i,N_j))=\begin{cases}\text{Hauteur de } N_j \text{ si } Etiquette(Arc(N_i,N_j))\in E_{opt}\\ 0 \text{ sinon}\end{cases}$$

Plus le coût d'un arc est élevé, plus la partie de la tâche correspondante a eu lieu tardivement, et inversement, plus le coût d'un arc est faible, plus elle a été exécutée précocement. Etant donné que tout mot du centre du langage terminal est un anagramme d'un autre, il y a exactement le même nombre d'arcs étiquetés par un élément de E_{opt} pour tout chemin de M_0 à M_f, donc un chemin aura un coût moins important qu'un autre si et seulement si l'ordonnancement correspondant exécute les parties des tâches de E_{opt} le plus tôt possible.

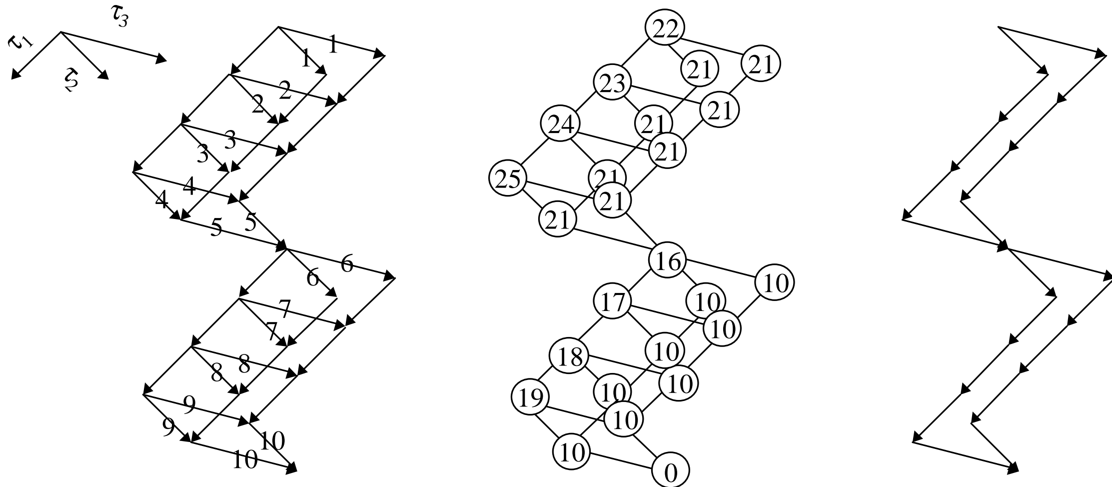


figure IV-2-11 : recherche des séquences de $S=\{\tau_1<0,3,5,5>, \tau_2<0,1,5,5>, \tau_3<0,2,10,10>\}$ *exécutant au plus tôt les tâches de* $E_{opt}=\{\tau_2, \tau_3\}$

La figure IV-2-11 illustre la recherche des séquences optimales exécutant au plus tôt τ_2 et τ_3. Le graphe de gauche montre le principe de pondération implicite des arcs. Elle est implicite car il n'est pas nécessaire d'effectuer un parcours préalable afin d'affecter les coûts puisqu'ils ne dépendent que de la hauteur des arcs reliés et de leur étiquetage. Le graphe central montre le résultat de la pondération des nœuds du graphe de telle sorte que le poids d'un nœud est le coût du plus court chemin partant de ce nœud et allant à M_f. Ce processus est basé sur un parcours topologique inverse (i.e. de bas en haut) du graphe, dont le principe est le suivant :

```
Coût(Nœud_Final)←0
Pour h=tc+P à 0 faire
      Pour tout nœud N de hauteur h faire
            Evaluer_Coût_Nœud(N,Ensemble_de_tâches)
      Fait
Fait
```

L'évaluation du coût d'un nœud se calcule simplement de la façon suivante : $Coût(N)=min_{N'\in fils(N)}\{Coût(N')+ Coût_AuPlusTôt(Arc(N,N'))\}$.

Chaque nœud et chaque arc est visité une et une seule fois, l'algorithme est donc linéaire dans la taille du graphe des marquages. Le graphe obtenu (voir graphe de droite dans la figure IV-2-11) est un sous-graphe du graphe entré dont tous les nœuds font partie d'un chemin de coût minimal.

IV-2.4.4.b Optimisation du temps de réponse

Le coût affecté à un arc correspondant à la terminaison de l'occurrence d'une tâche dont on veut optimiser le temps de réponse est le temps de réponse correspondant.

Cela s'exprime numériquement de la façon suivante : considérons deux marquages N_i et N_j reliés par l'arc $Arc(N_i,N_j)$ à la hauteur h, et étiqueté par une tâche τ_k. L'arc est pondéré par $Coût_TempsRéponse(Arc(N_i,N_j))=$

$$\begin{cases} 0 \text{ si } \tau_k \notin E_{opt} \text{ ou si l'arc ne correspond pas à une terminaison de } \tau_k \\ \text{sinon} \begin{cases} P_k \text{ si } h - r_k \equiv 0[P_k] \text{(i.e. si } \tau_k \text{ termine au moment d'être réactivée)} \\ h - \left\lfloor \dfrac{h - r_k}{P_k} \right\rfloor P_k - r_k \text{ sinon (temps de réponse de l'occurrence de } \tau_k \text{ à la hauteur } h\text{)} \end{cases} \end{cases}$$

Le coût d'un nœud est alors :
Coût(*N*)=*min*$_{N' \in fils(N)}$\{*Coût*(*N'*)+*Coût_TempsRéponse*(*Arc*(*N*,*N'*))\}.

figure IV-2-12 : minimisation du temps de réponse moyen des séquences d'ordonnancement d'un système S=\{τ_1<0,3,5,5>, τ_2<0,1,5,5>, τ_3<0,2,10,10>\} pour E_{opt}=\{τ_2, τ_3\}

La figure IV-2-12 illustre l'extraction de séquences minimisant le temps de réponse moyen d'un ensemble de tâches. Le graphe de gauche montre que seuls les arcs correspondant à une fin de tâche de E_{opt} sont pondérés. Le problème de ce critère est qu'il favorise la diminution du temps de réponse des tâches longues de E_{opt}. C'est encore plus flagrant lorsque l'on cherche à minimiser le temps de réponse maximal. La fonction de pondération des arcs est la même, mais le coût d'un nœud est alors *Coût*(*N*)=*min*$_{N' \in fils(N)}$\{*max*\{*Coût*(*N'*), *Coût_TempsRéponse*(*Arc*(*N*,*N'*))\}\}.

figure IV-2-13 : minimisation du temps de réponse maximal pour E_{opt}=\{τ_2, τ_3\}

La figure IV-2-13 illustre la minimisation du temps de réponse maximal d'un ensemble de tâches. Ce critère favorise la minimisation du temps de réponse des plus longues tâches au détriment des autres, puisque la première occurrence de τ_2 n'est pas minimisée du tout (voir figure IV-2-13).

Le critère taux de réaction est une alternative à ce phénomène.

IV-2.4.4.c Optimisation du taux de réaction

Le principe d'extraction de séquences optimisant le taux de réaction est identique au principe de minimisation du temps de réponse, cependant, dans ce cas, les coûts sont des rationnels.

$Coût_TauxRéaction(Arc(N_i,N_j)) = \dfrac{Coût_TempsRéponse(Arc(N_i, N_j))}{D_k}$ avec $Arc(N_i,N_j)$ étiqueté par τ_k.

Afin de minimiser le taux de réaction moyen, le coût d'un nœud est donné par $Coût(N)=min_{N' \in fils(N)}\{Coût(N')+Coût_TauxRéaction(Arc(N,N'))\}$. L'application de ce critère à un exemple est donné sur la figure IV-2-14.

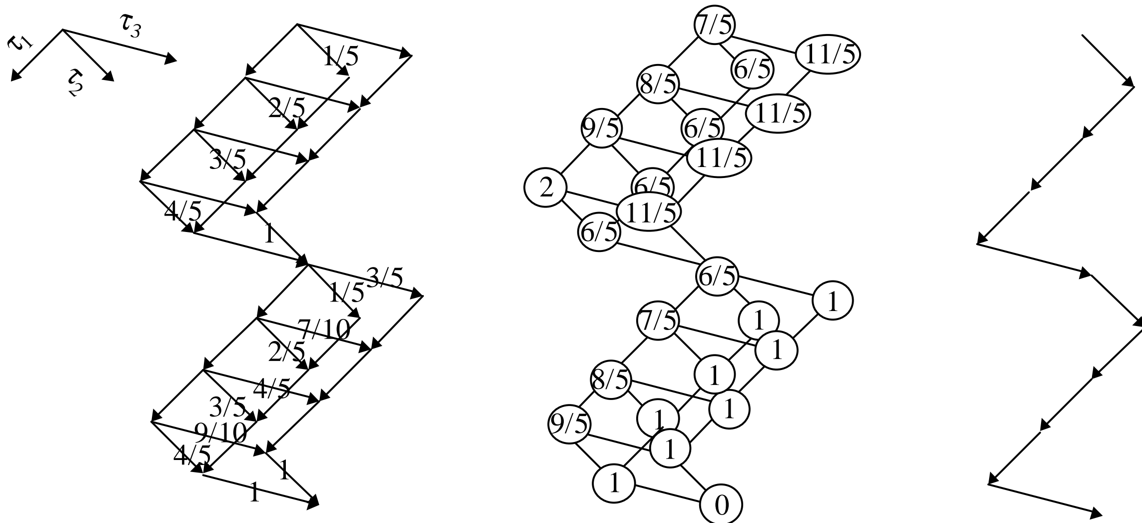


figure IV-2-14 : minimisation du taux de réaction moyen dans $S=\{\tau_1<0,3,5,5>, \tau_2<0,1,5,5>, \tau_3<0,2,10,10>\}$ pour $E_{opt}=\{\tau_2, \tau_3\}$

Pour minimiser le taux de réaction maximal, le coût d'un nœud sera donné par $Coût(N)=min_{N' \in fils(N)}\{max\{Coût(N'), Coût_TauxRéaction(Arc(N,N'))\}\}$. L'application de ce critère à l'exemple utilisé jusqu'ici ne réduit pas le graphe (voir figure IV-2-15), car pour toute séquence, au moins l'une des tâches de E_{opt} a un taux de réaction de 1.

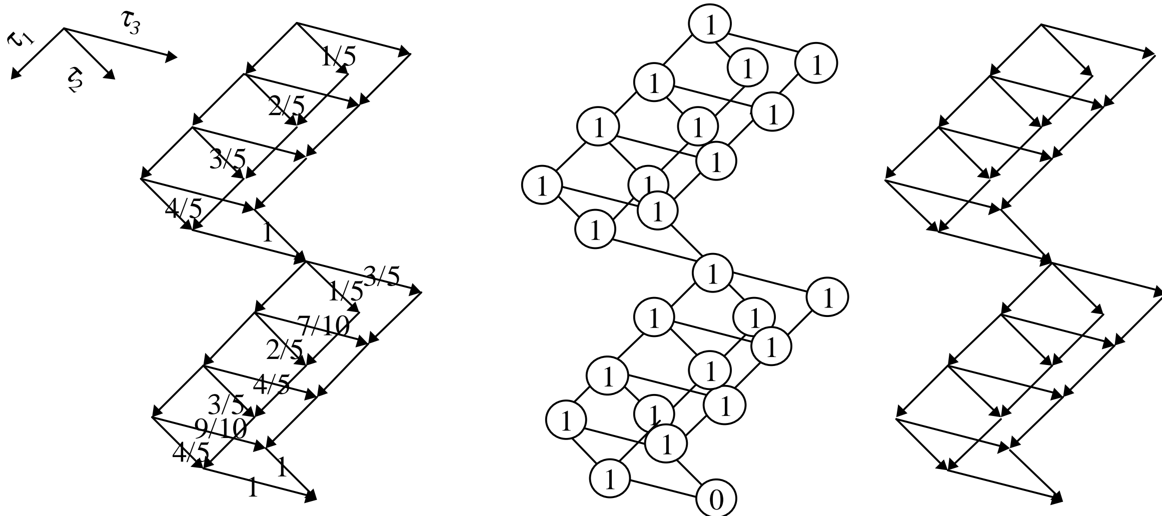


figure IV-2-15 : minimisation du taux de réaction maximal dans S={τ_1<0,3,5,5>, τ_2<0,1,5,5>, τ_3<0,2,10,10>} pour E_{opt}={τ_2, τ_3}

Le taux de réaction est donné par le temps de réponse divisé par le délai critique. Cependant, dans ce cas, les tâches les plus urgentes (de plus petit délai critique) sont favorisées.

IV-2.4.4.d Optimisation du retard

Un critère permettant de ne favoriser aucun type de tâche en particulier est le retard. Le retard d'une tâche est donné par son temps de réponse moins son délai critique. Dans le cas de séquences valides, le retard est toujours négatif ou nul, et nous nous proposons de trouver les séquences qui minimisent ce retard.

Coût_Retard(*Arc*(N_i,N_j))=*Coût_TempsRéponse*(*Arc*(N_i,N_j))–D_k avec *Arc*(N_i,N_j) étiqueté par τ_k.

Afin de minimiser le retard moyen, le coût d'un nœud est donné par *Coût*(*N*)=$min_{N' \in fils(N)}${*Coût*(*N'*)+*Coût_Retard*(*Arc*(*N*,*N'*))}. L'application de ce critère au même exemple que précédemment est donné sur la figure IV-2-16.

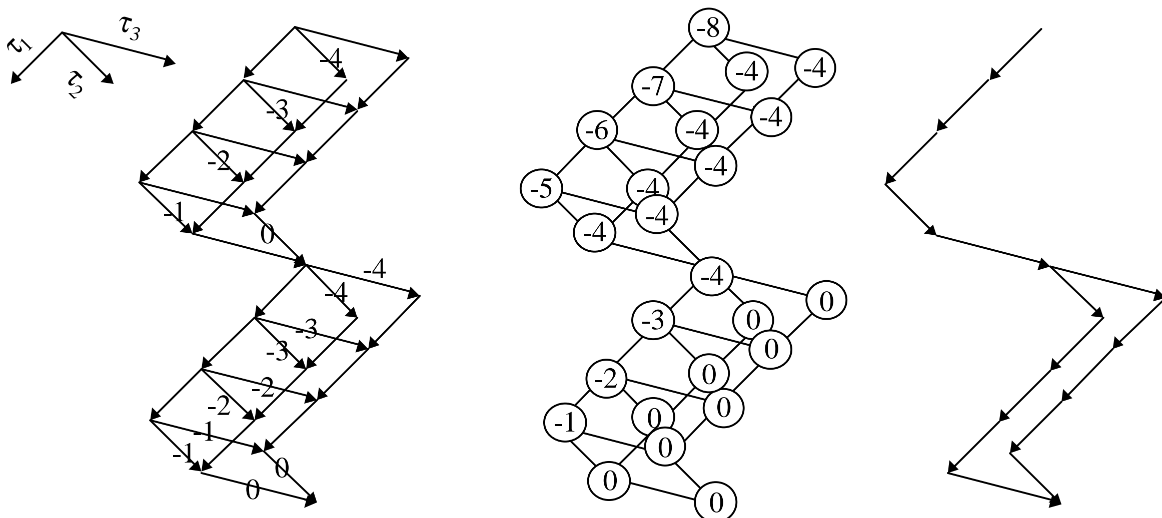


figure IV-2-16 : minimisation du retard moyen dans S={τ_1<0,3,5,5>, τ_2<0,1,5,5>, τ_3<0,2,10,10>} pour E_{opt}={τ_2, τ_3}

Le retard maximal est minimisé de la même façon que le temps de réponse maximal ou le taux de réaction maximal par la pondération :

$Coût(N)=min_{N' \in fils(N)}\{max\{Coût(N'), Coût_Retard(Arc(N,N'))\}\}$

Sur notre exemple, ce critère est aussi inefficace que la minimisation du taux de réaction maximal, en effet, pour toute séquence, soit τ_2 soit τ_3 a un retard de 0 (ce qui correspond à un taux de réaction de 1).

IV-2.4.4.e Minimisation de la gigue

Peu d'approches permettent à un concepteur de minimiser la gigue de certaines tâches sur l'ordonnancement produit, surtout si les tâches considérées ne sont pas de durée unitaire.

Une méthode qui peut être envisagée est la diminution du délai critique des tâches dont on veut minimiser la gigue, accompagnée d'un choix judicieux de leurs dates de réveil. La diminution du délai critique d'une tâche est souvent rédhibitoire pour les approches en-ligne, les conditions suffisantes d'ordonnançabilité étant trop strictement liées au ratio C_i/D_i dans le cas où les tâches ne sont pas à échéance sur requête. Dans le cas d'une approche hors-ligne, le choix d'un délai critique minimal ($C_i=D_i$) n'est pas forcément nuisible à l'ordonnançabilité du système considéré, du moment que les dates de réveil des tâches sont choisies de façon à ce que les tâches urgentes ne soient jamais en concurrence. Cette approche est souvent possible, puisque les paramètres temporels tels que la date de réveil et le délai critique des tâches ne sont généralement pas immuables dans le cahier des charges, et plus ou moins laissés à l'appréciation du concepteur. Il peut donc être intéressant de le guider dans ses choix [CCH 98]. [Dav 98] s'est intéressé aux tâches de durée unitaire, nous généralisons l'approche au cas de tâches de durée arbitraire.

Définition IV-2-3 : Une tâche urgente τ_i caractérisée temporellement par $<r_i,C_i,D_i,P_i>$ est telle que $C_i=D_i$.

Définition IV-2-4 : Une tâche urgente $\tau_i<r_i,C_i,D_i=C_i,P_i>$ est concurrencée par une tâche urgente $\tau_j<r_j,C_j,D_j=C_j,P_j>$ si et seulement si $\exists k \in I\!N$ et $c_i \in 0..C_i-1$ tels que $r_i+kP_i+c_i=\left\lceil \frac{r_i+kP_i-r_j}{P_j} \right\rceil P_j + r_j$.

Cela correspond au cas où τ_i ne peut pas être exécutée entièrement avant le réveil de τ_j car l'écart entre la date de réveil d'une occurrence de τ_i et celle de l'occurrence immédiatement suivante de τ_j est inférieur à C_i.

Le principe de concurrence entre deux tâches est illustré sur la figure IV-2-17.

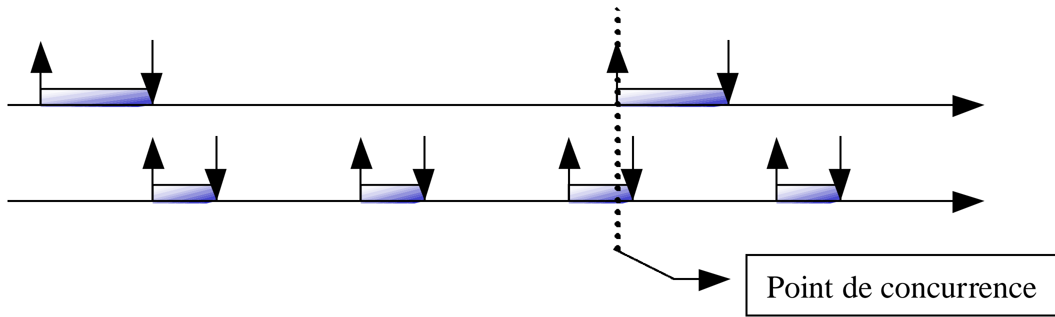


figure IV-2-17 : Deux tâche urgentes mises en concurrence

Lemme IV-2-1 :Le fait que les tâches urgentes ne rentrent jamais en concurrence deux à deux est une condition nécessaire d'ordonnançabilité.

Preuve : La preuve est triviale et vient du fait que si deux tâches urgentes sont mises en concurrence, on ne peut traiter les 2 à temps pour respecter leurs échéances respectives.

❏

Théorème IV-2-3 : Soit *S* un système de tâches composé en partie de tâches urgentes. Le fait que celles-ci ne rentrent jamais en concurrence deux à deux est une condition nécessaire et suffisante de la suppression de la gigue des tâches urgentes dans les ordonnancements valides.

Preuve : Condition suffisante : les tâches étant urgentes, elles doivent être exécutées dès leur activation et ont donc toujours un temps de réponse égal à leur charge si l'ordonnancement est valide.

Condition nécessaire : d'après le Lemme IV-2-1, la condition est nécessaire à l'ordonnançabilité du système.

❏

Théorème IV-2-4 : Une tâche urgente $\tau_i<r_i,1,1,P_i>$ est concurrencée par une tâche urgente $\tau_j<r_j,1,1,P_j>$ si et seulement si $r_i \equiv r_j\ [pgcd(P_i,P_j)]$.

Preuve : τ_j concurrence τ_i si et seulement si, par définition, $\exists k \in \mathbb{N}$, $r_i+kP_i=\left\lceil \frac{r_i+kP_i-r_j}{P_j} \right\rceil P_j + r_j$, ce qui signifie qu'il existe un entier x tel que $x \equiv r_i[P_i]$ et $x \equiv r_j[P_j]$ (1).

Théorème IV-2-5 : Théorème du reste chinois généralisé (*TRCG*) [Knu 81]. Soient p_1, $p_2,\ldots,p_m$ des entiers positifs, et r_1, r_2, …, r_m des entiers. Il existe exactement un entier x, tel que pour tout entier r, $r \le x < r+ppcm(p_i)_{i=1..m}$, $x \equiv r_j\ [p_j]$ $1 \le j \le m$ si et seulement si $r_i \equiv r_j\ [pgcd(p_i,p_j)]$, $1 \le i < j \le m$.

Après application du *TRCG* sur (1), on obtient le théorème.

❏

Il reste maintenant à généraliser le résultat à deux tâches urgentes de durée non unitaire.

Théorème IV-2-6 : Une tâche urgente τ_i<$r_i,C_i,D_i=C_i,P_i$> n'est pas concurrencée par une tâche urgente τ_j<$r_j,C_j,D_j=C_j,P_j$> si et seulement si $C_i<pgcd(P_i,P_j)$ et $(r_j-r_i)[pgcd(P_i,P_j)]\geq C_i$.

<u>Preuve :</u> τ_j concurrence τ_i si et seulement si, par définition, $\exists k\in \mathbb{N}$ et $c_i\in 0..C_i-1$ tels que $r_i+kP_i+c_i=\left\lceil \frac{r_i+kP_i-r_j}{P_j} \right\rceil P_j+r_j$, ce qui signifie qu'il existe un entier x tel que { $x\equiv r_i[P_i]$ ou $x\equiv(r_i+1)[P_i]$ ou … ou $x\equiv(r_i+C_i-1)[P_i]$ } et $x\equiv r_j[P_j]$ (i.e. τ_j est activée dans un intervalle $[r_i+kP_i..r_i+kP_i+C_i-1]$). Après application du *TRCG*, on obtient la formule équivalente { $r_i\equiv r_j[pgcd(P_i,P_j)]$ ou $r_i+1\equiv r_j[pgcd(P_i,P_j)]$ ou … ou $r_i+C_i-1\equiv r_j[pgcd(P_i,P_j)]$ }

τ_i n'est donc pas concurrencée par τ_j si et seulement si $\{(r_i\neq r_j)[pgcd(P_i,P_j)]$ et $(r_i+1\neq r_j)[pgcd(P_i,P_j)]$ et… et $r_i+C_i-1\neq r_j[pgcd(P_i,P_j)]\}$. Donc on peut écrire de façon équivalente $\{(r_j-r_i\neq 0)[pgcd(P_i,P_j)]$ et $(r_j-r_i\neq 1)[pgcd(P_i,P_j)]$ et… et $r_j-r_i\neq C_i-1[pgcd(P_i,P_j)]\}$. Plaçons sur la figure IV-2-18 les valeurs interdites de r_j-r_i dans $\mathbb{Z}/p\mathbb{Z}$, avec $p=pgcd(P_i,P_j)$, et observons les valeurs restantes.

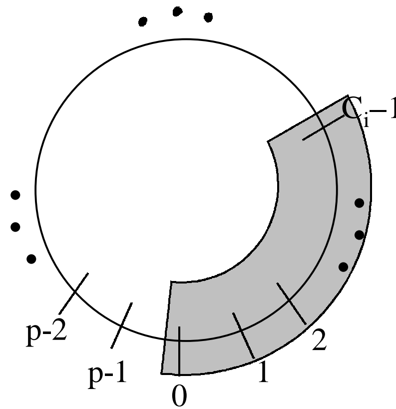


figure IV-2-18 : valeurs interdites de r_j-r_i dans $\mathbb{Z}/p\mathbb{Z}$ avec $p=pgcd(P_i,P_j)$

Sur cette figure, les valeurs interdites pour r_j-r_i sont grisées, les valeurs possibles sont le complémentaires dans $\mathbb{Z}/p\mathbb{Z}$ de l'ensemble des valeurs grisées. On peut remarquer qu'il est nécessaire et suffisant que $C_i<pgcd(P_i,P_j)$ pour que cet ensemble soit non vide, car dans le cas contraire, toutes les valeurs de $\mathbb{Z}/p\mathbb{Z}$ seraient grisées. Les valeurs autorisées donc l'ensemble que l'on peut caractériser dans $\mathbb{Z}/p\mathbb{Z}$ par $(r_j-r_i)\geq C_i[pgcd(P_i,P_j)]$.

❏

Ce théorème est un outil permettant de choisir la date de réveil des tâches urgentes de façon à ce qu'elles ne soient jamais concurrentes. Lorsque le concepteur doit choisir la date de réveil de deux tâches urgentes τ_i et τ_j, il doit donc utiliser ce théorème dans les deux sens, puisqu'il ne faut pas que τ_i soit concurrencée par τ_j ni que τ_j soit concurrencée par τ_i. Cela limite donc, d'après le théorème, le choix de la différence $(r_j-r_j)[pgcd(P_i,P_j)]$ dans la partie non grisée de la figure IV-2-19.

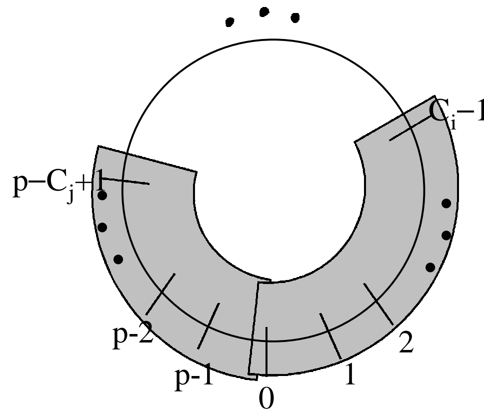


figure IV-2-19 : les valeurs possibles pour le choix de (r_j−r_i) pour éviter leur concurrence, avec p=pgcd(P_i,P_j)

> Théorème IV-2-7 : Soit *S* un système de tâches urgentes :
>
> 1. *S* est ordonnançable si et seulement si aucune tâche n'est concurrente.
>
> 2. Tout ordonnancement valide est de gigue nulle pour l'ensemble des tâches.

Preuve :

1. Sens du si : si les tâches ne sont jamais concurrentes deux à deux, alors elles peuvent toutes s'exécuter de façon non préemptive sans être gênées par aucune autre tâche. Sens du seulement si : découle du Lemme IV-2-1.
2. Découle du Théorème IV-2-3.

❏

Ce théorème s'utilise de la façon suivante : pour tester l'ordonnançabilité d'un système de tâches urgentes indépendantes, il suffit de vérifier pour tout couple de tâches τ_i et τ_j que τ_i ne concurrence pas τ_j et vice-versa. Si les tâches sont soumises à des contraintes de précédence, il faut de plus vérifier la cohérence des précédences (les occurrences des tâches émettrices doivent avoir une date de réveil antérieure aux occurrences des tâches réceptrices associées). Les ressources critiques sont ignorées car une tâche urgente ne peut être préemptée.

Cependant, le cas de systèmes de tâches toutes urgentes est peu réaliste, et dans le cas des systèmes de tâches mixtes (dans le sens où certaines sont urgentes et d'autres non), le Théorème IV-2-7 appliqué aux tâches urgentes n'est plus qu'une condition nécessaire d'ordonnançabilité.

Dans le cadre de notre approche, il n'est pas obligatoire de rendre les tâches urgentes pour les exécuter au plus tôt, surtout lorsque cette contrainte rend le système non ordonnançable, mais on peut a posteriori exiger les séquences qui minimisent le temps de réponse des tâches dont on veut minimiser la gigue. Le concepteur aura, si possible, ajusté les dates de réveil des tâches dont il veut minimiser la gigue en tenant compte du Théorème IV-2-6.

Le problème de cette méthode est qu'elle n'est pas exacte. En effet, habituellement on oblige les tâches dont on veut éliminer la gigue à s'exécuter dès leur activation, ce qui peut être rédhibitoire à l'ordonnançabilité. Une autre approche consisterait à étudier le graphe

d'ordonnancement afin d'en extraire les séquences de gigue minimale pour une tâche donnée, si possible de façon similaire à l'extraction de séquences à temps de réponse optimal.

Soit τ_i la tâche dont on veut minimiser la gigue, décomposée en ses $n_i=\left\lfloor \frac{t_c+P+1-r_i}{P_i} \right\rfloor$ instances respectives, formant un ensemble noté $E'_{opt}=\{\tau_{i_1}, \tau_{i_2},\ldots, \tau_{i_{ni}}\}$, d'échéances respectives $\{d_{i_1}, d_{i_2},\ldots, d_{i_{ni}}\}$, de temps de réponse respectifs $\{tr_{i_1}, tr_{i_2},\ldots, tr_{i_{ni}}\}$. Minimiser la gigue de τ_i revient à minimiser l'écart des temps de réponse entre ses différentes occurrences, i.e. minimiser l'écart entre $tr_{i_1}, tr_{i_2},\ldots, tr_{i_n}$. Le problème est que l'écart entre des temps de réponse dépend non pas de la position d'un arc mais de ses passés et de ses avenirs (dans le cas de la minimisation du temps de réponse, le coût d'un arc est indépendant de l'histoire des séquences passées et futures). Afin d'illustrer ce problème, la figure IV-2-20 représente les séquences d'ordonnancement valides pour un système de tâches $S=\{\tau_1<0,1,2,2>, \tau_2<0,1,4,4>\}$, et les arcs en pointillés représentent les *2* séquences de gigue nulle. Ce graphe, comportant *4* séquences possibles, n'en possède que *2* de gigue nulle. Si l'on souhaite représenter le sous-graphe ne contenant que des séquences optimisant la gigue, il faut éclater le nœud N_3 en deux nœuds, ce qui fait grossir le graphe des marquages, et, à terme, peut faire perdre le bénéfice d'une représentation sous forme de graphe par rapport à une représentation arborescente.

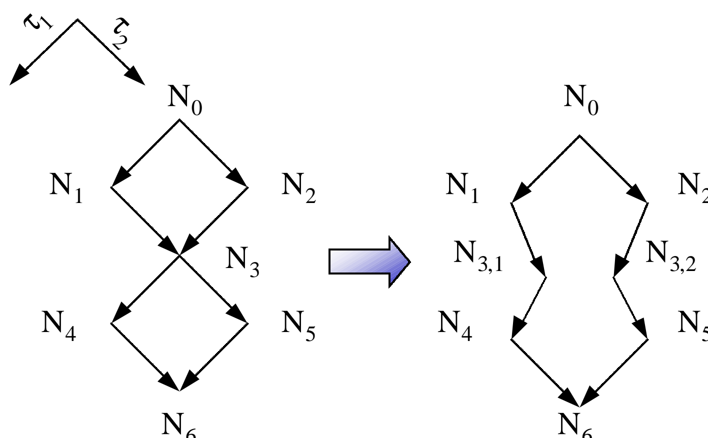


figure IV-2-20 : ordonnancements de deux tâches, deux séquences sont de gigue nulle

En effet, un suffixe ne peut être optimal en regard de la gigue qu'en fonction de son préfixe et vice-versa. Nous parlons alors de critère arborescent, puisque son optimisation nécessite l'éclatement du graphe en arbre. Il nous faut donc aborder ce problème sous un angle plus limité. Nous décomposons le problème en la recherche d'une gigue minimale successivement pour chacun des temps de réponse possibles de τ_i.

Pour chaque temps de réponse possible tr de C_i à D_i, la pondération d'un arc correspondant à la terminaison d'une occurrence de τ_i est donnée par la distance entre sa date d'occurrence et tr. Ainsi, un chemin de coût minimal est un chemin pour lequel les occurrences de la tâche considérée ont eu lieu le plus près possible de tr. Il reste à choisir le temps de réponse offrant un coût minimal et à conserver le sous-graphe minimisant la gigue des tâches par rapport à cette date. En cas d'ex-æquo, un choix doit être fait afin de conserver la structure de graphe et de ne pas avoir à éclater des nœuds.

Le coût d'un arc est la distance entre le temps de réponse choisi tr et le temps de réponse de la tâche τ_k dont on veut minimiser la gigue :

Coût_Gigue(*Arc*(N_i,N_j),*tr*)=|*Coût_TempsRéponse*(*Arc*(N_i,N_j))−*tr*| où |*a*| est la valeur absolue de *a*.

Le coût d'un nœud *N*, donnant le coût du chemin minimisant la somme des distances entre temps de réponse de τ_k et le *tr* choisi est alors :

Coût(*N*)=$min_{N'\in fils(N)}$\{*Coût*(*N'*)+*Coût_Gigue*(*Arc*(*N*,*N'*),*tr*)\}

Les chemins les plus courts minimisent la gigue.

La figure IV-2-21 montre le graphe des marquages correspondant à la minimisation de la gigue de la tâche τ_1 pour un temps de réponse le plus proche possible de *3*.

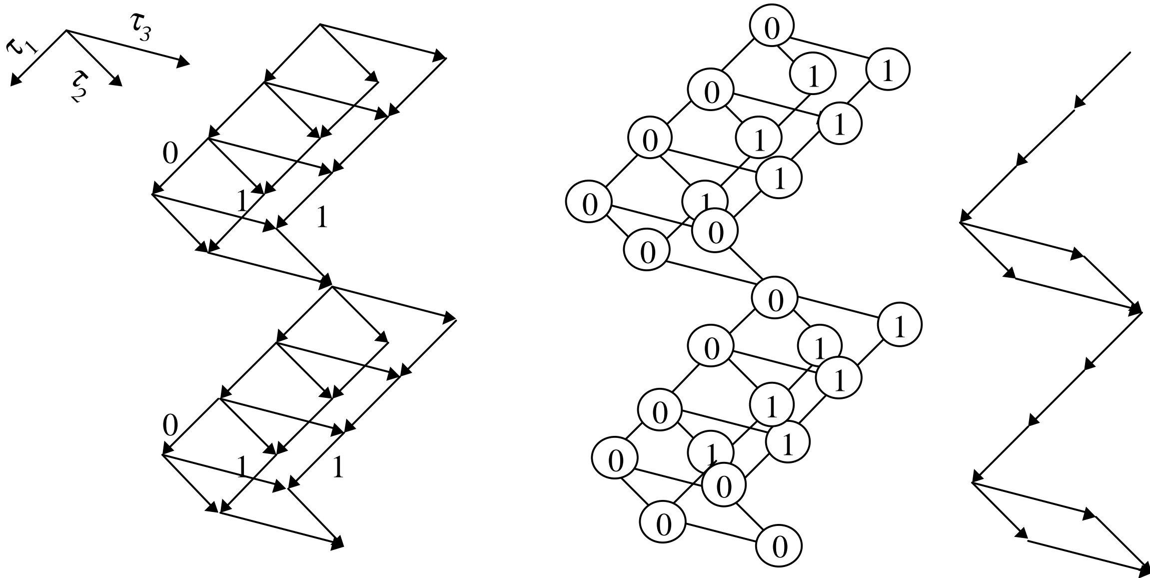


figure IV-2-21 : minimisation de la gigue de τ_1 *dans* $S=\{\tau_1<0,3,5,5>, \tau_2<0,1,5,5>, \tau_3<0,2,10,10>\}$ *avec un temps de réponse proche de 3*

Le nœud N_0 étant étiqueté par *0*, il existe des chemins pour lesquels le temps de réponse de τ_1 est toujours *3*. Il est à noter que nous aurions pu choisir de conserver le sous-graphe minimisant la gigue de τ_1 par rapport au temps de réponse *4* qui aurait eu la même gigue nulle.

L'algorithme d'obtention d'une gigue minimale nécessite jusqu'à D_i-C_i+1 recherches de plus courts chemins dans le graphe. De plus seul un sous ensemble de séquences optimales est choisi en cas d'ex-æquo entre différents temps de réponse. Enfin, ce critère n'est appliqué que sur une tâche à la fois. Cependant, c'est la seule approche, à notre connaissance, qui permet de minimiser de façon systématique la gigue d'une tâche à paramètres temporels fixes (voir le début du paragraphe pour une méthode basée sur la modification des dates de réveils).

IV-2.4.4.f Répartition des temps creux

Le critère répartition des temps creux consiste à choisir les séquences dans lesquelles les temps creux sont équitablement répartis tout au long de la séquence. En effet, avec les séquences d'ordonnancement généralement produites, le concepteur minimise petit à petit le temps de réponse des tâches qu'il juge importantes, et finalement, les temps creux sont la plupart du temps relégués en fin de cycle d'ordonnancement. Dans le cas fréquent où le système est en fait mixte et qu'il y a des tâches de fond à traiter en même temps que le système, le fait de leur allouer une fenêtre assez large à des intervalles de temps espacés augmente le temps de réponse moyen face à des (relativement courtes) requêtes apériodiques de tâches molles.

La répartition des temps creux permet de pallier cet inconvénient en offrant de façon équilibrée des temps creux tout au long du cycle de la séquence produite.

Le principe d'obtention de séquences optimales est le même que celui permettant de minimiser la gigue d'une tâche, la répartition étant un critère arborescent.

Rappelons que le graphe des marquages est composé d'un premier diamant allant de la hauteur 0 à t_c+1, correspondant à la montée en charge du système, puis d'un second diamant allant de t_c+1 à t_c+P+1 correspondant à la partie cyclique des ordonnancements. Nous ne nous intéressons qu'à la répartition des temps creux cycliques, c'est à dire à ceux qui font partie du diamant cyclique. Une seule occurrence de la tâche oisive est donc présente dans le diamant considéré puisque τ_0 démarre à la date t_c+1 et a une période de P. Notre but est donc de répartir chacun des $C_0=P(1-U_S)$ temps creux équitablement entre t_c+1 et t_c+P+1. Nous choisissons donc, comme pour la minimisation de la gigue, une première date voulue, tr, d'occurrence du premier temps creux cyclique, comprise entre t_c+1 et t_c+1+P/C_0, puis nous cherchons à effectuer le deuxième temps creux le plus près possible de $tr+P/C_0$, ..., le $i^{ème}$ le plus près possible de $tr+(i-1)P/C_0$ etc…

On a donc, pour un arc de hauteur h correspondant au $i^{ème}$ temps creux :

$Coût_Répartition(Arc(N_i,N_j),tr)=|h-tr-(i-1)P/C_0|$

Et $Coût_Répartition(Arc(N_i,N_j),tr)=0$ dans les autres cas.

Le coût minimal d'un suffixe passant par un nœud $Coût(N)$ est donc :

$Coût(N)=min_{N'\in fils(N)}\{Coût(N')+Coût_Répartition(Arc(N,N'),tr)\}$

La figure IV-2-22 illustre sur un exemple l'optimisation de la répartition des temps creux.

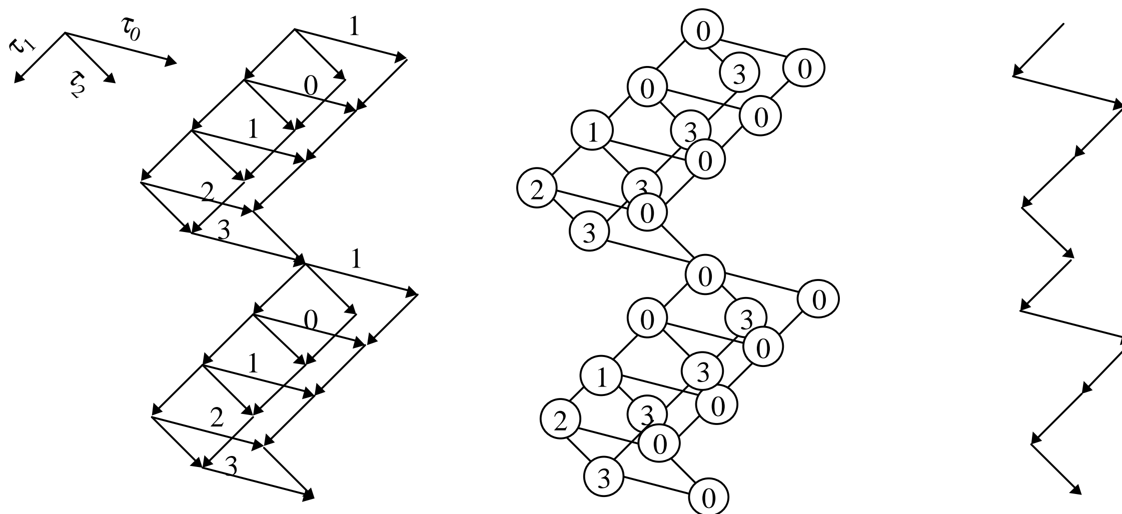


figure IV-2-22 : optimisation dans S={τ_1<0,3,5,5>, τ_2<0,1,5,5>, τ_3<0,2,10,10>}de la répartition des temps creux pour un premier temps de réponse de 2

De la même façon que pour la minimisation de la gigue, la répartition d'une tâche ne nous permet pas de conserver toutes les séquences optimales, nous nous limitons donc au graphe dont la répartition a été ajustée à partir d'une première date donnée.

IV-2.4.4.g Discussion sur les critères qualitatifs

L'avantage des critères qualitatifs est qu'ils fournissent un outil puissant de sélection de séquences a posteriori dans un graphe d'ordonnancement. Il est possible de raffiner le *GM* en appliquant successivement plusieurs critères et ainsi d'obtenir un ensemble réduit de séquences optimisant un certain nombre de critères. Des limites s'imposent cependant sur les critères

arborescents (gigue, répartition), nous contraignant à abandonner des séquences optimales. De plus, il faut garder en mémoire le fait que, étant donné des fonctions de raffinement f_1 et f_2, $f_1°f_2 \neq f_2°f_1$ dans le cas général : en effet, appliquer f_1 sur le graphe raffiné par f_2 signifie que les séquences obtenues sont celles qui optimisent le critère ciblé par f_1 parmi les séquences qui optimisent le critère f_2.

A notre connaissance, un outil permettant de tels choix de séquences est inédit.

Cependant, les critères qualitatifs ont un inconvénient majeur : étant donnée leur nature (ce sont des critères relatifs), ils ne peuvent être utilisés qu'après construction exhaustive du graphe des marquages. Il peut donc être utile de les coupler à un ensemble de contraintes a priori (absolues ou de successeur) qui diminueront d'emblée le graphe des marquages. Les contraintes choisies doivent être compatibles de préférence avec le critère à optimiser : par exemple les critères arborescents sont incompatibles avec les contraintes de successeur.

V Etude de Systèmes Temps Réel à l'Aide de Réseaux de Petri: Cas Multiprocesseur

V-1 Hypothèses matérielles

Nous nous plaçons dans le cadre du modèle *PRAM* dans lequel chaque processeur est identique et a une mémoire privée, mais une mémoire commune est accessible en temps constant par tous les processeurs.

Les communications ne sont pas caractérisées par un coût différent suivant le processeur choisi. Cela peut être un coût donné (par exemple le coût d'accès à la mémoire centrale) pour tout processeur exactement comme pour le cas monoprocesseur : le coût d'un message est pris en compte avant émission dans le corps de la tâche émettrice.

Rappelons qu'une tâche est la plus petite entité parallélisable d'un programme, c'est à dire qu'une tâche ne peut pas être traitée simultanément sur plusieurs processeurs.

On peut envisager un placement des tâches a priori. Cependant, ce choix est relativement rare dans le modèle *PRAM* puisque la symétrie entre les processeurs est totale.

Par contre en ce qui concerne la **migration**[12] des tâches, différentes hypothèses peuvent être formulées. En effet, on peut imaginer des architectures sur lesquelles les tâches migrent sans aucune contrainte, d'autres où une requête de tâche commencée sur un processeur s'y exécute entièrement, et d'autres enfin, où les tâches s'exécutent toujours sur le même processeur. L'hypothèse la plus répandue est la première, dans laquelle les tâches peuvent être déplacées d'un processeur à un autre en cours d'exécution. Dans ce cas, nous considérons que les changements de contexte des tâches passent par la mémoire commune.

Définition V-1-1 : Les hypothèses possibles de migration des tâches sont les suivantes :

- L'**absence de migration**, notée $\boldsymbol{M^0}$: aucune tâche n'est autorisée à migrer, le processeur choisi par la première occurrence de la tâche sera par la suite toujours choisi.
- La **migration faible**, notée $\boldsymbol{M^1}$: chaque occurrence d'une tâche peut avoir lieu sur un processeur différent, mais une occurrence de tâche ne peut pas migrer.
- La **migration forte,** notée $\boldsymbol{M^2}$: chaque occurrence d'une tâche peut changer de processeur à tout moment.

Les hypothèses M^0 et M^1 sont plus réalistes dans le cadre d'une architecture distribuée, alors que l'hypothèse M^2 l'est plus dans l'hypothèse multiprocesseur C'est donc cette hypothèse que nous avons choisi d'étudier de façon détaillée.

[12] On dit d'une tâche qu'elle migre si son exécution a lieu sur différents processeurs, de façon, bien sûr, non concurrente.

V-2 Modélisation

Le modèle tel qu'il était présenté dans le cadre de l'environnement monoprocesseur est particulièrement bien adapté à la modélisation de systèmes de tâches du contexte $|m|r_i,C_i,D_i,P_i$, *prmptnoprmpt*, *prec*, *resRW*| sous l'architecture M^2. En effet, chaque tâche, via les transitions modélisant son corps, concourrait avec les autres pour l'obtention de la ressource processeur modélisée par un jeton.

La séquentialité des transitions représentant les tâches garantit naturellement qu'une tâche ne peut pas s'exécuter en même temps sur plusieurs processeurs. Cependant, contrairement au cas monoprocesseur, la tâche oisive peut s'exécuter parallèlement sur plusieurs processeurs. On ne peut donc plus modéliser les temps creux cycliques par une seule tâche mais par un ensemble de tâches unitaires : ainsi, plusieurs processeurs peuvent en même temps être oisifs.

Le modèle est donc identique au modèle utilisé dans le cadre monoprocesseur sauf en ce qui concerne les points suivants :

- la place processeur contient initialement m jetons au lieu d'un, m étant le nombre de processeurs disponibles pour le système de tâches,
- la tâche oisive est remplacée par $P(m-U_S)$ tâches oisives de durée unitaire et de période et de délai critique P. Cependant, ce remplacement étant coûteux en transitions pour le modèle, une autre solution est de paralléliser les transitions représentant les temps creux cycliques au lieu de les sérialiser comme pour une tâche classique. Mais à ce moment là, $P(m-U_S)$ transitions seraient en conflit pour m processeurs. Imaginons par exemple qu'il y ait *7* temps creux représentés par *7* transitions $T_{0,1,1},\ldots, T_{0,7,1}$ mises en parallèle et *2* processeurs. Le fait que les deux processeurs soient oisifs serait représenté par le tir de n'importe quel couple pris dans les transitions de τ_0, ce qui représente $\binom{7}{2}$=21 possibilités représentant la même action : laisser les deux processeurs oisifs. Dans le cas général, ce modèle engendrerait $\binom{P(m-U_S)}{m}$ possibilités de tir différentes pour représenter le fait que les m processeurs sont oisifs. Ce modèle ne serait donc pas efficace dans une approche comme la nôtre se basant sur le graphe des marquages. La modélisation retenue est donc celle qui est représentée sur l'exemple suivant.

exemple V-2-1 : un système de 3 tâches intéragissantes sur *2* processeurs.

```
Tâche τ1 est
r1 = 0 ;
P1 = 8 ;
D1 = 8 ;
Début
  b1,1=2 ;
  Déposer_BAL(MB) ;
  b1,2=2 ;
Fin ;
```

```
Tâche τ2 est
r2 = 0 ;
P2 = 8 ;
D2 = 8 ;
Début
  Retirer_BAL(MB) ;
  b2,1=1 ;
  Prendre_Sémaphore(R,1) ;
  b2,2=4 ;
  Rendre_Sémaphore(R,1) ;
Fin ;
```

```
Tâche τ3 est
r3 = 0 ;
```

```
P3 = 16 ;
D3 = 16 ;
Début
  b3,1=1 ;
  Prendre_Sémaphore(R,1) ;
  b3,2=2 ;
  Rendre_Sémaphore(R,1) ;
  b3,3=1 ;
Fin ;
```

La charge du système est U_S=*11/8*, donc il y a *10* temps creux par méta-période.

Les marquages terminaux sont données par $\Psi=\{M\in(\mathbb{N}\times\{a\}\times\{b\})^{|Q|} \mid M(time_i)=1\Rightarrow M(activ_i)=\{a,b\}$ ou $\{b\}\}_{i=0..3}$, et le modèle est représenté sur la figure V-2-1. La tâche oisive est représentée par m (ici $m=2$) transitions parallèles mises en exclusion (une seule peut être tirée à la fois). A chaque méta-période, $P(m-U_S)$ jetons sont produits dans $activ_0$, chaque jeton représente un temps creux qui a lieu dans une méta-période. Le tir de la transition $T_{0,1,i}$ pour i allant de 1 à m, signifie que i processeurs sont oisifs : i jetons sont consommés dans $activ_0$ et i jetons processeur sont utilisés. Comme dans le cas monoprocesseur, chaque transition $T_{i,j,k}$ des tâches autres que τ_0 utilise une ressource processeur, mais nous ne le représentons pas pour alléger le graphique.

De plus, la transition $T_{0,1,1}$ est étiquetée par le singleton $\{\tau_0\}$, $T_{0,1,2}$ est étiquetée par la paire $\{\tau_0,\tau_0\}$,…, et la transition $T_{0,1,m}$ est étiquetée par le multi-ensemble $\{\tau_0\}^m$ (on note $\{a\}^m$ le multi-ensemble formé de m occurrences de la lettre a).

En résume, le modèle est en tout point identique au modèle monoprocesseur, sauf que la place processeur contient initialement m jetons, et que la tâche oisive est modélisée de la façon suivante :

- Son activation se traduit par la production de $P(m-U_S)$ jetons dans $activ_0$,
- Son corps est un ensemble de m transitions mises en exclusion mutuelle à l'aide d'une place ressource $exclusif_0$. Pour $i=1..m$, la transition $T_{0,1,i}$ consomme i jetons dans $activ_0$ en utilisant i jetons processeur.

L'étiquetage de ses transitions est tel que la transition $T_{0,1,i}$ est le multi-ensemble $\{\tau_0\}^i$.

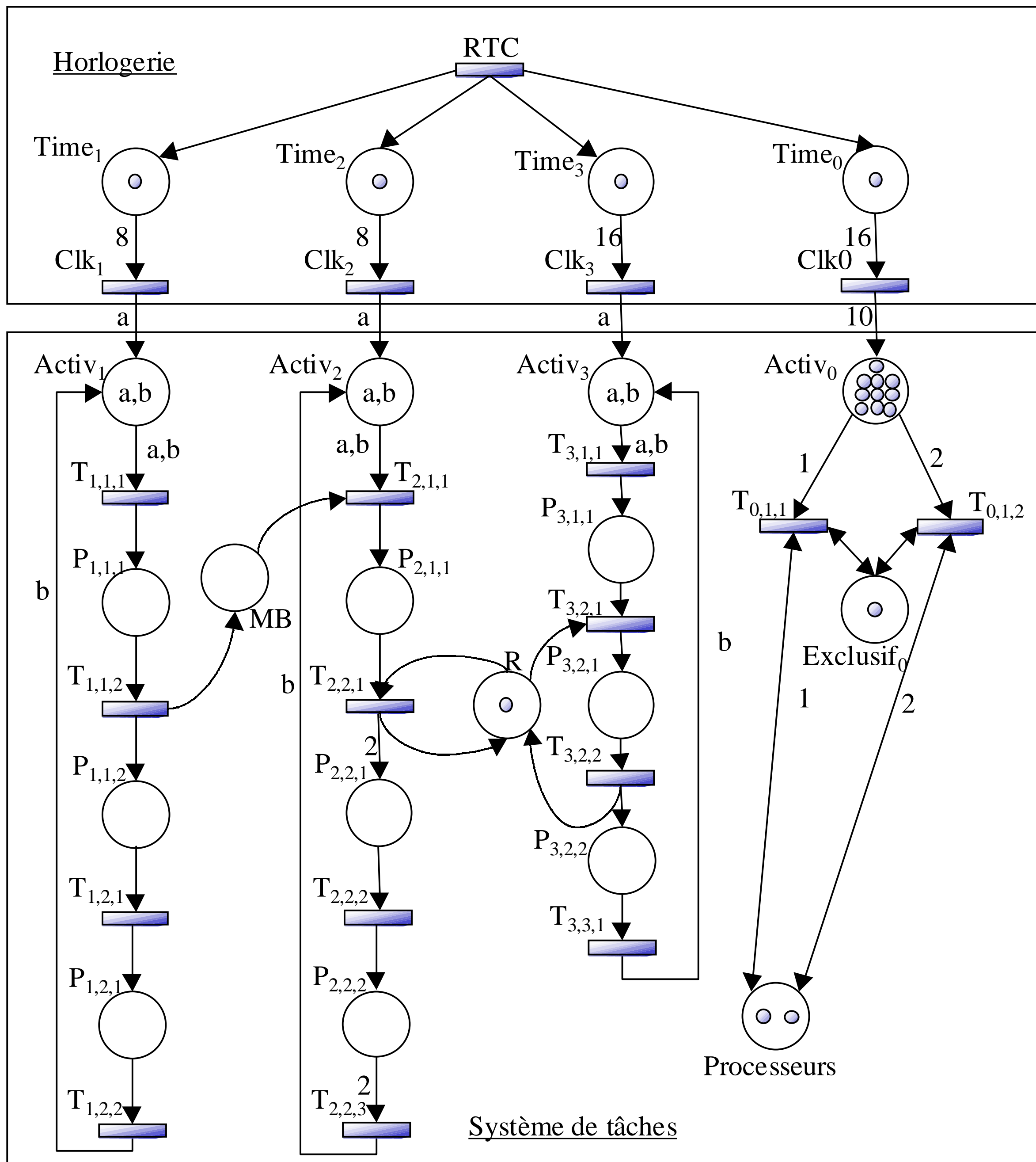


figure V-2-1 : modélisation d'un système de tâches interagissantes à l'aide d'un RdP. On ne représente pas graphiquement le lien qui existe entre le processeur et chaque transition du système de tâches (i.e. $W(Processeur,T_{i,j,k})=W(T_{i,j,k},Processeur)=1$)

V-3 Etude

Nous nous limitons aux systèmes de tâches simultanées, car le résultat de cyclicité des ordonnancements de tâches différées n'a été établi que dans le cas monoprocesseur.

Comme dans le cas monoprocesseur, le centre du langage terminal du modèle est l'ensemble des séquences d'ordonnancement valides du système de tâche modélisé. Cependant, dans le cas monoprocesseur, chaque arc du graphe des marquages est étiqueté par un singleton $\{\tau_i\}$ qui signifie que la tâche τ_i a progressé. Dans le cas multiprocesseur, étant donné que m transitions du système de tâches peuvent être franchies en parallèle chaque arc du *GM* est étiqueté par un multi-ensemble (en effet τ_0 peut être présent plusieurs fois) de cardinal m.

V-3.1 Profondeur du graphe d'accessibilité

Cette section décrit l'étude du graphe des marquages correspondant à la modélisation des systèmes de tâches simultanées. Il est tout à fait possible d'étudier le cas des systèmes de tâches différées : il suffit pour cela de simuler le modèle afin de construire le graphe des marquages. Chaque marquage inséré après la hauteur P est alors comparé aux marquages se trouvant P niveaux plus haut dans le graphe (on sait étant donné l'existence des places horloges locales qu'il est inutile de comparer des marquages n'ayant pas été obtenus à la même date modulo P). Puisque le RdP est borné, le graphe des marquages l'est aussi. Mais les techniques de recherche de séquences optimales que nous utilisons s'appuient sur une forme de diamant du graphe des marquages. Donc si le *GM* n'a pas cette forme particulière, son étude est nettement plus délicate.

D'après le Théorème I-2-3, chaque séquence d'ordonnancement du contexte $|p,m|r_i=0,C_i,D_i,T_i,$ *prmptnoprmpt*,*prec*,*resRW*| est cyclique de période P à partir de l'instant 0. Donc le *GM* a la même forme de diamant que dans le cas monoprocesseur, et chaque chemin a une longueur P.

V-3.2 Représentation du graphe d'accessibilité

Dans le cas des systèmes de tâches simultanées, il n'y a pas de temps creux acyclique, donc chaque arc du *GM* est étiqueté par un multi-ensemble de cardinal m, $\{\tau_{i,1}, \tau_{i,2,\dots,} \tau_{i,m}\}$, signifiant que les tâches $\tau_{i,j}$ ont été traitées.

Afin d'illustrer les études à venir, nous donnons deux exemples de *GM* obtenus dans le cas multiprocesseur.

V-3.2.1 Deux tâches simultanées indépendantes de charge 100%

exemple V-3-1 : Soit $S=\{\tau_1<0,3,3,3>, \tau_2<0,6,6,6>\}$ un système de tâches indépendantes ordonnancées sur 2 processeurs. La charge du système est $U_S=2$, c'est à dire que *100%* du temps de deux processeurs sera utilisé par S. Il n'est donc pas nécessaire d'ajouter une tâche oisive. Le graphe des marquages produit par le modèle RdP de S est donné sur la figure V-3-1.

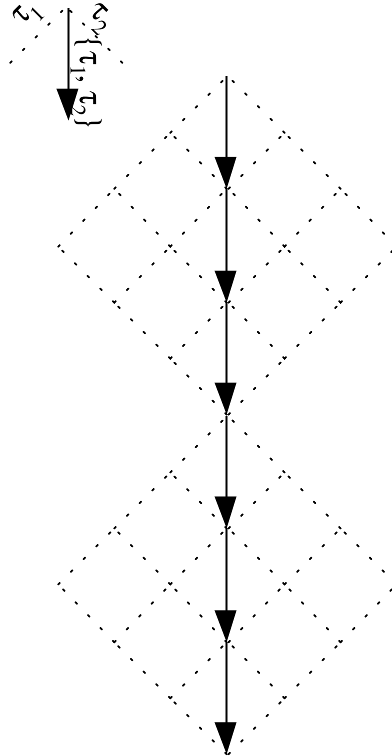


figure V-3-1 : un graphe d'accessibilité contenant l'ensemble des séquences d'ordonnancement valides d'un système de tâches de charge maximale sur 2 processeurs

Sur cette figure, on voit qu'il n'y a qu'une seule séquence d'ordonnancement de S, qui est $\sigma = \{\tau_1,\tau_2\}\{\tau_1,\tau_2\}\{\tau_1,\tau_2\}\{\tau_1,\tau_2\}\{\tau_1,\tau_2\}\{\tau_1,\tau_2\}$. Chaque tâche τ_i est répétée $\frac{mP}{P_i}$ fois en P unités de temps, donc l'ensemble des processeurs doit lui consacrer $\frac{mPC_i}{P_i}$ unités de temps. Les pointillés représentent l'ensemble des états qu'il serait possible d'atteindre si l'on était en environnement monoprocesseur mais que chaque tir ne durait que $1/m$ unités de temps. La différence entre le graphe des marquages pour m processeurs et le graphe obtenu en pointillés réside d'une part dans le fait qu'au moins m unités de temps de traitement sont effectuées à chaque unité de temps, et d'autre part une tâche autre que la tâche oisive ne peut pas être traitée simultanément sur plusieurs processeurs.

V-3.2.2 Deux tâches simultanées indépendantes de charge inférieure à 100%

---

exemple V-3-2 : Soit $S=\{\tau_1<0,6,6,6>,\ \tau_2<0,5,6,6>\}$ un système de tâches indépendantes ordonnancées sur 2 processeurs. La charge du système est $U_S=11/6$, c'est à dire qu'il faut ajouter au système une tâche oisive $\tau_0<0,1,6,6>$ afin d'augmenter artificiellement la charge du système jusqu'à *100%* du temps de deux processeurs. Le graphe des marquages produit par le modèle RdP de S est donné sur la figure V-3-2.

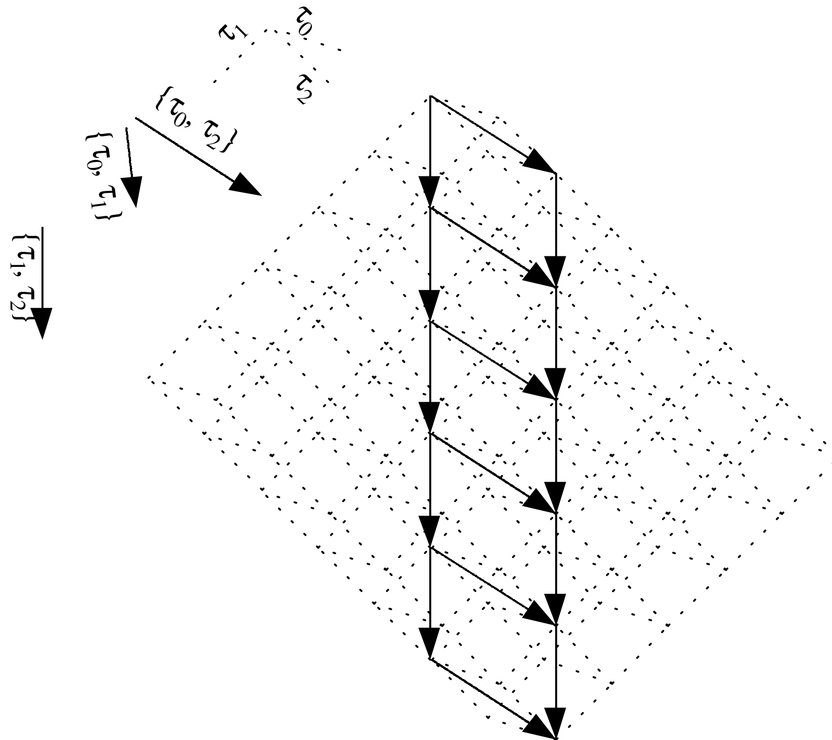


figure V-3-2 : un graphe d'accessibilité contenant l'ensemble des séquences d'ordonnancement valides d'un système de tâches sur 2 processeurs

Sur cette figure, il y a *6* séquences d'ordonnancement valides. Comme dans le cas précédent, chaque arc du *GM* correspond à une diagonale du *GM* correspondant au cas monoprocesseur dans lequel chaque unité de temps vaut $1/m$ unités de temps.

V-3.3 Taille du graphe d'accessibilité

Il est clair que les sommets du graphe des marquages sont tous inclus dans les sommets de l'hyperpavé que l'on obtiendrait en considérant le cas rêvé d'un environnement monoprocesseur dans lequel le processeur traite *m* quanta de temps par unité de temps. Cela correspond graphiquement aux graphes donnés en pointillés sur la figure V-3-1 et la figure V-3-2. On obtient donc une borne supérieure du nombre des sommets en considérant le nombre de sommets obtenus dans un hyperpavé de dimension le nombre de tâches, et de cotes respectives $\frac{mPC_i}{P_i}$. Ce qui donne une borne de $(mP)^{n+1}\prod_{i=0}^{n}\frac{C_i}{P_i}$ sommets pour *n* tâches plus la tâche oisive. Cette borne est relativement large, puisque d'une part, comme nous l'avons vu pour le cas monoprocesseur, les contraintes temporelles et structurelles coupent l'hyperpavé théorique. De plus, le fait d'obliger *m* tâches à progresser à chaque instant ampute encore le graphe qui serait obtenu sur un processeur capable de traiter *m* unités de temps par unité de temps.

V-3.4 Obtention des séquences d'ordonnancement valides

V-3.4.1 Placement des tâches a posteriori

Comme dans le cas monoprocesseur, un chemin du *GM* est une séquence d'ordonnancement valide du système modélisé. Cependant, aucune information n'est donnée par le modèle sur le placement des tâches, puisque les processeurs sont des jetons non différenciés. En effet, l'information donnée par l'étude d'un arc du *GM* pour un système de *3* processeurs peut être : les tâches τ_1, τ_2 et τ_5 sont exécutées. Rien ne dit que τ_1 s'exécute sur le processeur 1, τ_2 sur le processeur *2* et τ_5 sur le processeur *3*. Cela évite les symétries qui apparaîtraient si les processeurs étaient différenciés, comme par exemple les événements suivants : {τ_1 s'exécute sur le processeur 1, τ_2 s'exécute sur le processeur *2*} et {τ_1 s'exécute sur le processeur *2*, τ_2 s'exécute sur le processeur 1}.

Le placement n'est pas donné par le modèle mais obtenu après à partir de la séquence des actions fournie par le modèle. En effet, une séquence d'ordonnancement est, dans le cas de tâches simultanées auquel nous nous intéressons, un mot $\sigma = w_1w_2 \ldots w_P$ où $w_{k,\ k=1..P}$ est un multi-ensemble de cardinal *m*. Pour $t \in \{1,..,P\}$, $w_t=\{\tau_{i,1},\ldots, \tau_{i,m}\}$ signifie que les tâches $\{\tau_{i,1},\ldots, \tau_{i,m}\}$ se voient attribuer un processeur pour un quantum de temps à l'instant *t*.

Connaissant la séquence d'ordonnancement, il reste à placer les tâches sur les processeurs afin d'éviter les migrations et préemptions inutiles. On pourrait aussi tenter de minimiser les messages inter-processeurs, mais dans la mesure où, par hypothèse, les communications inter-processeurs ne sont pas plus coûteuses que les communications n'impliquant qu'un seul processeur, nous négligeons volontairement ce problème dans la présente étude.

Nous nous intéressons donc à la minimisation des changements de contexte. Il s'agit d'assurer qu'une tâche ne change pas de processeur lorsqu'elle travaille en continu : en effet, alors qu'une tâche préemptée nécessite de toutes façons un changement de contexte, une tâche que l'on peut ne pas préempter ne devra pas nécessiter de changement de contexte, donc tant qu'une tâche n'est pas préemptée, elle doit rester sur le même processeur. L'algorithme de placement, utilisé après la construction du *GM*, est donné en annexe I-5. Il permet d'éviter les changements de contexte inutiles en évitant de déplacer une tâche qui est exécutée de façon continue.

Cependant, même lorsqu'une tâche est préemptée, il peut être intéressant de lui allouer le même processeur lorsqu'elle est à nouveau active, pour profiter de la mémoire cache.

On peut donc améliorer cet algorithme en se basant sur une approche de type *LRU* (*Least Recently Used*) : lorsqu'une tâche se voit ré-allouer un processeur, le fait de travailler sur le même permet d'aller chercher son contexte dans le cache et d'éviter d'avoir à faire transiter le contexte par la mémoire commune.

Etant donné que les coûts de communication inter-processeurs sont négligés, ainsi que les coûts de migration, le placement est fait uniquement lorsque l'utilisateur choisit d'afficher une séquence après raffinages successifs afin d'obtenir des séquences optimales. En effet, le graphe des marquages est exempt d'informations sur le placement puisque celui-ci est une propriété arborescente (i.e. pour un arc donné *Arc*(*M*,*M'*) c'est le chemin choisi du nœud M_0 à cet arc qui induit le placement, et non pas l'arc lui-même).

V-3.4.2 Diminution des cas de retours arrière

Dans le cas monoprocesseur, nous utilisons une technique de détection précoce des fautes temporelles permettant de diminuer sensiblement le nombre de retours arrières effectués lors

de la construction du *GM* qui, rappelons-le, ne doit contenir que des mots infinis du centre du langage terminal du modèle RdP.

Cette technique s'applique sur chaque nœud : la latence de chaque tâche, c'est à dire sa marge de manœuvre, est évaluée et permet de déterminer de façon précoce le fait qu'un nœud ne peut mener qu'à des fautes temporelles. Elle se base sur la condition nécessaire d'ordonnançabilité de $n+1$ tâches sur un processeur qui est $\forall k \in 0..n$, $\sum_{i=0}^{k} C_i(t) \leq L_k(t)$ où $C_i(t)$ est la charge restant à traiter pour terminer la tâche τ_i et $L_i(t)$ est sa latence à l'instant t.

Sur m processeurs, cette CS devient $\forall k \in 0..n$, $\sum_{i=0}^{k} \frac{C_i(t)}{m} \leq L_k(t)$. Elle peut être affinée, puisque si une tâche, en dehors de la tâche oisive, n'a pas le temps de s'exécuter entièrement sur un seul processeur, alors il y aura une faute temporelle. En effet, les tâches (mis à part τ_0) ne pouvant être exécutées simultanément sur plusieurs processeurs, il est intéressant de vérifier ce fait à chaque nœud du *GM* lors de sa construction. Cela se traduit par la CS supplémentaire $\forall i \in 1..n$, $C_i(t) \leq L_i(t)$.

L'algorithme de vérification de la première CS s'effectue comme pour la CS monoprocesseur en un temps $\theta(n)$ par nœud, et la complexité de la seconde CS est la même.

V-3.4.3 Note sur les contraintes de successeur

Dans le cas monoprocesseur, nous utilisons des contraintes de successeur, heuristique diminuant les changements de contexte inutiles en terme d'ordonnançabilité. Pour rappel, les contraintes interdisent à une tâche d'en préempter une autre sauf dans les cas suivants :

1. Une tâche qui vient d'être (ré)activée peut préempter la tâche travaillant
2. Une tâche qui vient d'être débloquée par la réception d'un message ou la libération d'une ressource peut préempter la tâche travaillant
3. Si la tâche travaillant demande son entrée en section critique, toute tâche peut la préempter.

Ces contraintes ne peuvent pas être utilisées dans le cas multiprocesseur car il peut être indispensable pour trouver une séquence valide, de faire que des tâches indépendantes se préemptent mutuellement. Cela est dû au fait qu'une tâche ne peut pas s'exécuter simultanément sur plusieurs processeurs. Ce fait est illustré par l'exemple V-3-3.

---

exemple V-3-3 : Soit un système de *3* tâches indépendantes ordonnancées sur deux processeurs $S=\{\tau_1<0,2,3,3>,\tau_2<0,2,3,3>,\tau_3<0,2,3,3>\}$. Bien que les tâches soient indépendantes et réveillées simultanément, l'obtention d'une séquence d'ordonnancement valide nécessite au moins une préemption. La figure V-3-3 (a) donne une séquence valide comportant une préemption, et la figure V-3-3 (b) représente la séquence obtenue (modulo les permutations des noms de tâches puisqu'elles sont toutes identiques) en utilisant les contraintes de successeur, c'est à dire dans ce cas, en interdisant toute préemption.

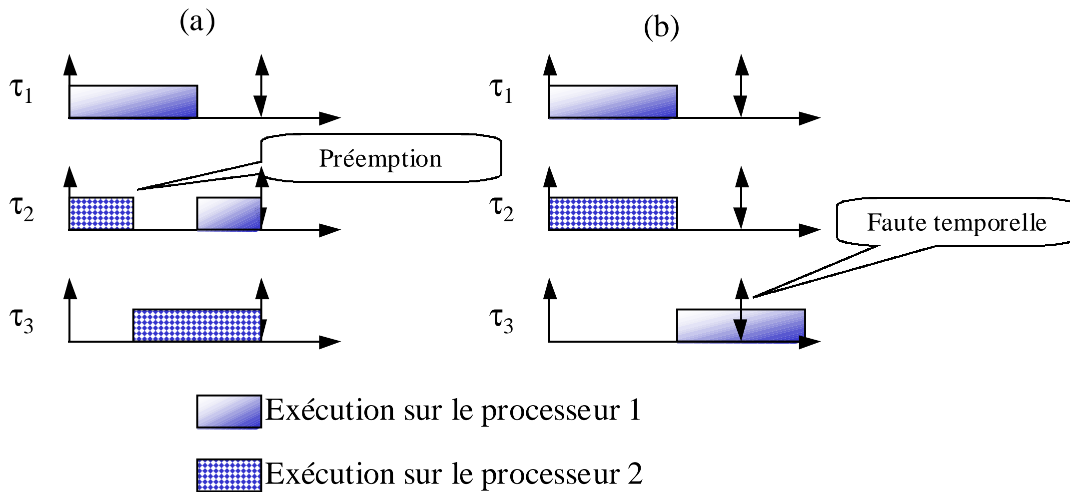


figure V-3-3 : séquences d'ordonnancement montrant que les préemptions entre tâches indépendantes sont nécessaires à l'ordonnançabilité d'un système

V-3.4.4 Diminution de la taille du graphe des marquages

Les contraintes absolues peuvent être utilisées sans modification majeure dans le cas multiprocesseur. Rappelons qu'elles sont basées sur un calcul le plus fin possible du meilleur temps de réponse possible d'une tâche à partir d'un nœud *N* du *GM*.

Comme précédemment, l'algorithme de calcul du temps de réponse minimal utilise la table des latences croissantes fabriquée pour dépister de façon précoce la non validité d'un nœud.

```
Fonction TempsDeRéponseMinimalMulti(N:Noeud, τi:nom_de_tâche) Renvoie entier
      Min,t : entier
      τj,indice : nom_de_tâche
Début
      t←hauteur(N)
      Min←Cτi(t)
      -- Charge restant à traiter pour τj
      indice←première tâche de la table des latences croissantes
      Tant que (Min+Cindice(t))/m>Lindice(t) Faire
            -- m est le nombre de processeurs
            -- Lindice(t) est la latence de la tâche indice
            Min←Min+Cindice(t)
            indice←tâche suivante dans la table
      Fait
      Renvoyer Min+t-max(0,⌊(t-ri)/Pi⌋Pi)
Fin
```

V-3.4.5 Extraction de séquences d'ordonnancement optimales

Les techniques d'extraction de séquences d'ordonnancement optimales sur les critères exécution au plus tôt, temps de réponse, taux de réaction et retard minimaux présentées dans le cadre monoprocesseur s'appliquent sans modification au cas multiprocesseur.

La seule nuance est que le coût d'un arc, puisqu'il est étiqueté dans le cas multiprocesseur par un multi-ensemble de cardinal m, est la somme des coûts qui seraient affectés à chaque élément du multi-ensemble.

De la même façon, les algorithmes de recherche de séquences assurant un répartition homogène des temps creux ou minimisant la gigue peuvent être appliqués tels qu'ils ont été définis dans le cas monoprocesseur.

VI Application de l'Etude

VI-1 Présentation de *PeNSMARTS*

Puisque nous utilisons une technique exponentielle en temps et en espace au pire, il nous a paru indispensable de mettre en œuvre une maquette permettant de montrer, sur des exemples concrets, la faisabilité de notre étude. Le logiciel résultant a pour nom *PeNSMARTS* pour *Petri Net Scheduling, Modeling and Analysis of Real-Time Systems*.

Le propos de ce chapitre est la présentation de cet outil, et des fonctionnalités que nous y avons implémentées. Dans sa forme actuelle, il n'intègre que la partie monoprocesseur de notre étude.

PeNSMARTS s'intègre dans le projet *MOSARTS* du LISI, qui a motivé à ce jour deux études de doctorat (thèses de J.P. Babau [Bab 96][BC 98] et S. Saad-Bouzefrane [Saa 98][SC 99]). C'est un atelier d'étude et de validation d'applications temps réel. La première thèse étudie les algorithmes en-ligne classiques (*ED*, *DM*, *LL*,…) dans le cadre monoprocesseur, et la seconde applique ces techniques aux environnements répartis. Ce troisième volet, incarné par *PeNSMARTS*, permet la création de séquences d'ordonnancement hors-ligne, notamment pour les systèmes temps réel très contraints en terme de ressources.

VI-1.1 Saisie du système de tâches à étudier

L'outil se base sur une grammaire proche de celle que nous avons utilisée pour donner les exemples de systèmes de tâches non indépendantes, afin, pour commencer, de construire le modèle RdP présenté dans la section I-1. Cette grammaire, dont l'axiome est *S*, est donnée ci-dessous, avec les mot clef en gras, et les symboles en lettres capitales, les commentaires étant mis, comme en C, entre /* et */. Notre souci a été de conserver une compatibilité avec les outils de *MOSARTS*, notre grammaire, sauf exception, est celle présentée dans [Bab 96][BC 98].

```
/* Axiome */
S →   APPLICATION ; NOMBREDETÂCHES; NOMBREDEMESSAGES; NOMBREDERESSOURCES;
      TÂCHE*
/* Nom de l'application */
APPLICATION → application CHAINE
/* Nombre de tâches */
NOMBREDETÂCHES → task number : ENTIER
/* Nombre de messages (de boîtes aux lettres) */
NOMBREDEMESSAGES → event number : ENTIER
/* Nombre de ressources */
NOMBREDERESSOURCES → resource number : ENTIER
/* Description d'une tâche via ses 3 paramètres temporels et son corps */
TÂCHE → task(CHAINE); ri : ENTIER; Ri : ENTIER;Ti : ENTIER; CORPSDETÂCHE
/* Corps d'une tâche */
CORPSDETÂCHE → begin INSTRUCTION* end
/* Les différentes instructions possibles */
INSTRUCTION →     d(ENTIER,ENTIER) = ENTIER;
                  /* L'entier après le signe = est la durée du bloc */
                  /* Les 2 autres sont conservés pour compatibilité avec*/
                  /* MOSARTS */
            |     lock(res(ENTIER)[,readonly]);
                  /* Chaque ressource est implicitement numérotée */
                  /* readonly est spécifique à PeNSMARTS afin d'exprimer */
                  /* l'utilisation d'une ressource en lecture */
            |     unlock(res(ENTIER));
            |     send(event(ENTIER));
```

```
                  /* Chaque boîte aux lettres est implicitement numérotée*/
            |     wait(event(ENTIER));
ENTIER → [0-9]+
CHAINE → [a-zA-Z_][[a-zA-Z_0-9]*
```

Il est important de constater que chaque tâche, dans cette syntaxe, est abstraite à sa durée et à ses primitives temps réel. Nous avons vu en section I-1 comment construire le modèle RdP correspondant.

Cependant, une grammaire textuelle étant assez rébarbative à saisir, nous avons développé en Tcl/Tk [Wel 97] une petite surcouche graphique *QTSD* pour *Quick Task System Designer* permettant une construction interactive d'un système de tâches et générant le code source adéquat. Une copie d'écran de *QTSD* est donnée sur la figure VI-1-1.

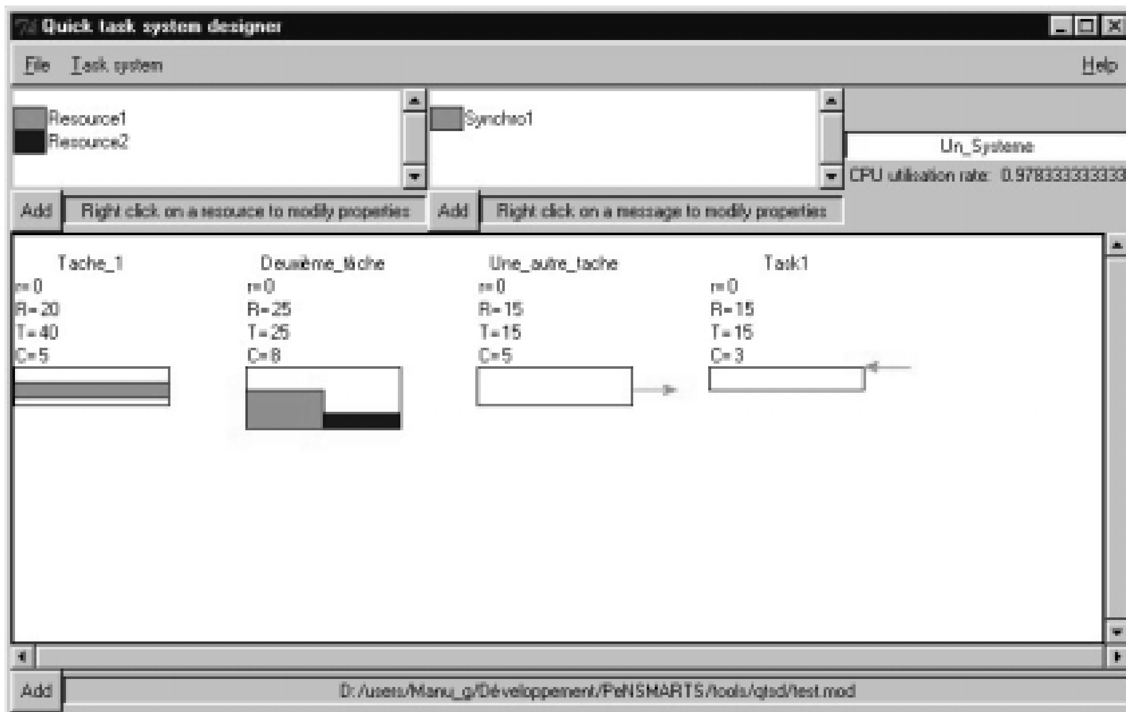


figure VI-1-1 : copie d'écran de QTSD

La fenêtre de *QTSD* est composée de *4* parties. En haut à gauche, la partie ressources permet de créer des ressources critiques, différenciées par des couleurs. En haut au centre, l'utilisateur peut créer des boîtes aux lettres. En haut à droite, un entrée de texte permet la saisie du nom de l'application, et la charge processeur est calculée automatiquement à l'aide des paramètres temporels des tâches.

La plus grande fenêtre représente les tâches, avec une boîte par tâche. La charge processeur (C_i) permet de régler la taille du rectangle représentant la durée de la tâche éditée. A l'aide de la souris, l'utilisateur peut prendre et libérer une ou plusieurs ressources, ce qui est représenté par des couleurs dans le rectangle visualisant le corps de la tâche concernée. Par exemple, sur la figure VI-1-1, la première tâche utilise le ressource 1 pendant deux unités de temps après un traitement de deux unités de temps, alors que la seconde tâche utilise la ressource 1 pendant *5* unités de temps après *3* unités de temps de traitement, et la ressource *2* pendant *2* unités de temps après *6* unités de temps de traitement.

Les envois et réceptions de message sont représentés graphiquement par des flèches de couleur, entrantes ou sortantes, comme c'est le cas pour les tâches *3* et *4*.

VI-1.2 Génération du modèle RdP

Une fois le système de tâches saisi, de façon textuelle ou graphique, l'utilisateur peut lancer *PeNSMARTS*. Après ouverture du fichier de description d'un système de tâches, le logiciel crée automatiquement le modèle RdP. Le RdP créé peut d'ailleurs être visualisé, à titre indicatif, à l'aide d'un joueur de jetons que nous avons développé lui aussi en Tcl/Tk. Une copie d'écran de cet outil, nommé *CPNS* pour *Colored Petri Net Simulator* est donnée sur la figure VI-1-2.

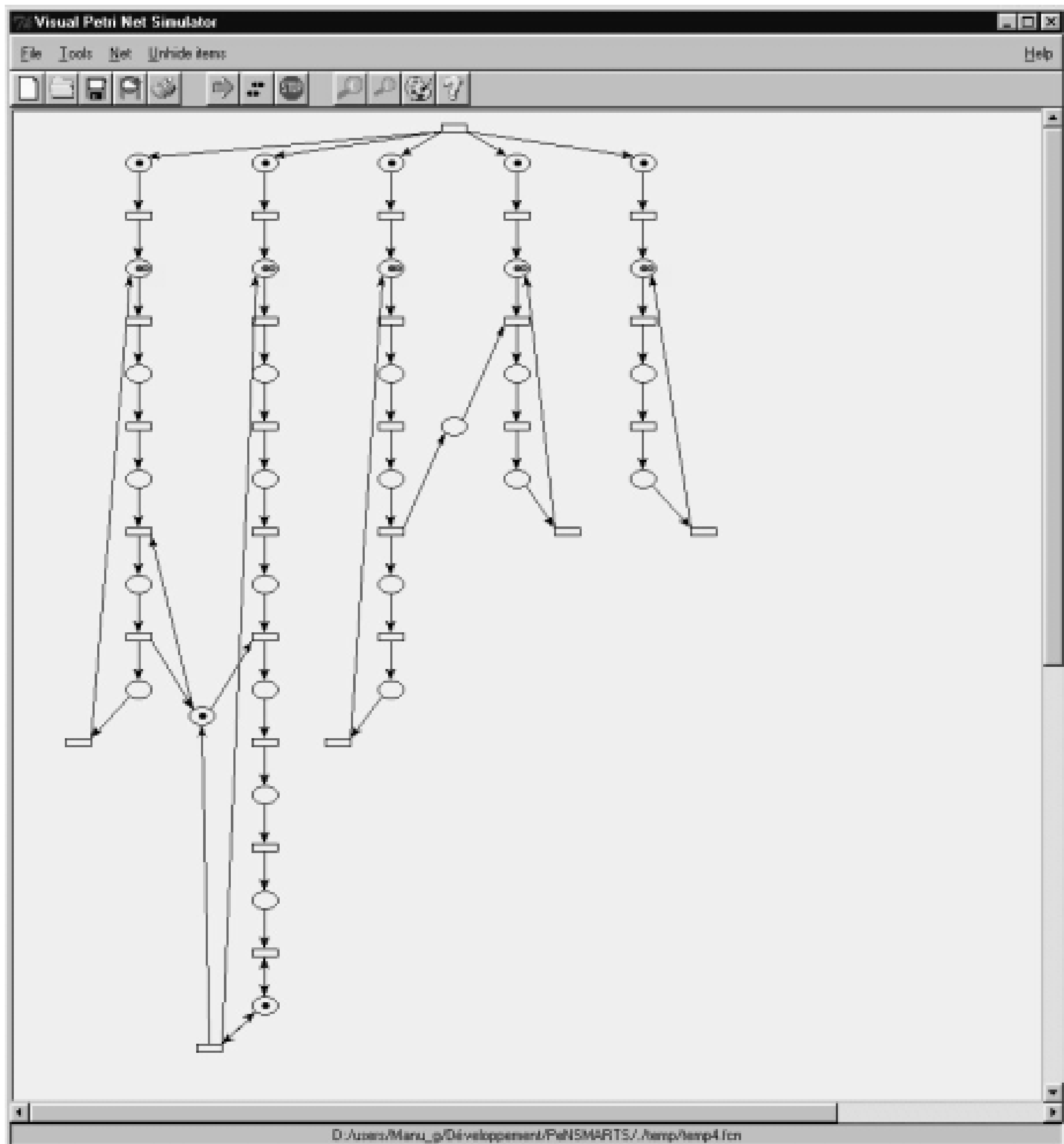


figure VI-1-2 : copie d'écran du joueur de jetons CPNS

VI-1.3 Génération des séquences d'ordonnacement

L'outil *PeNSMARTS* affiche une fenêtre reprenant alors les principales caractéristiques des tâches, enrichi automatiquement de la tâche oisive τ_0. La figure VI-1-3 représente ce que voit alors l'utilisateur.

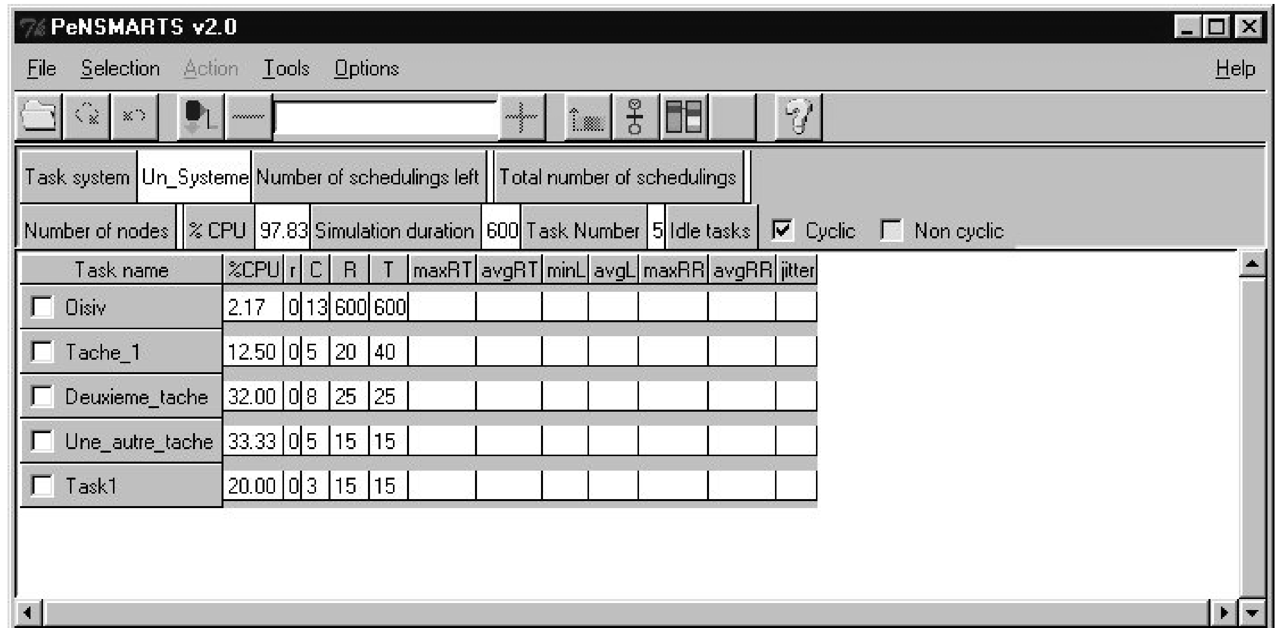


figure VI-1-3 : copie d'écran de PeNSMARTS

Cette figure montre que pour l'exemple considéré, une tâche oisive a été ajoutée, et que la durée de simulation nécessaire est de 600 unités de temps. La barre d'outils permet de lancer automatiquement les outils déjà présentés que sont *QTSD* et *CPNS*.

L'utilisateur est alors libre de choisir d'utiliser ou non les heuristiques de construction à l'aide des options présentes dans le menu *File*. Ces options, représentées sur la figure VI-1-4, permettent à l'utilisateur de choisir ou non d'utiliser les contraintes de successeurs définies en section IV-2.4.3.a, les contraintes absolues (voir section IV-2.4.3.b), et les optimisations de construction permettant de diminuer les retours arrière (voir section IV-2.4.2). Cette dernière optimisation est à utiliser dans tous les cas, puisqu'elle ne diminue en rien la puissance d'ordonnançabilité de l'outil, cependant il est possible de la désactiver afin de comparer la vitesse de construction avec et sans cette optimisation. Par exemple, pour l'application jouet utilisée ici, l'obtention du même *GM* nécessite sur un PENTIUM III à *450*MHz *8* secondes de traitement avec l'optimisation, et *11* secondes sans (calcul effectué avec l'utilisation de contraintes de successeur).

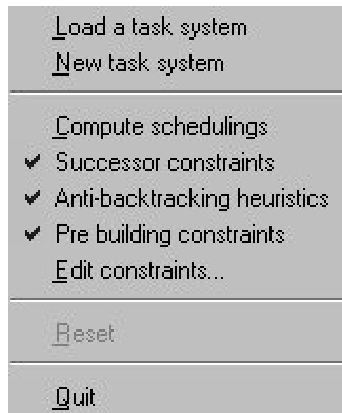


figure VI-1-4 : options disponibles pour la construction du graphe des marquages

Un clic sur l’icône en forme de moulinette permet de lancer la construction du graphe des marquages.

PeNSMARTS v2.0

File Selection Action Tools Options Help

Task system: Un_Systeme — Number of schedulings left: 63295673296777958767352676057624831 323936870233945940213967842929803264

Number of nodes: 9232 — % CPU: 97.83 — Simulation duration: 600 — Task Number: 5 — Idle tasks — Cyclic — Non cyclic

Task name	%CPU	r	C	R	T	maxRT	avgRT	minL	avgL	maxRR	avgRR	jitter
Oisiv	2.17	0	13	600	600	526	526	74	74	263/300	263/300	0
Tache_1	12.50	0	5	20	40	40	581/15	0	19/15	2	581/300	4
Deuxieme_tache	32.00	0	8	25	25	25	265/12	0	35/12	1	53/60	13
Une_autre_tache	33.33	0	5	15	15	15	75/8	0	45/8	1	5/8	10
Task1	20.00	0	3	15	15	13	67/8	2	53/8	13/15	67/120	7

figure VI-1-5 : PeNSMARTS après construction du graphe des marquages

PeNSMARTS présente alors de nouvelles informations, comme nous le voyons sur la figure VI-1-5 : le nombre de séquences d’ordonnancement valides, que l’on serait bien incapable de lire tellement il est grand, et le nombre de nœuds nécessaires à leur stockage, en l’occurrence *9232*. La boîte contenant à ce moment le chiffre 1 et encadrée par un bouton « + » et un bouton « - » permet de choisir une séquence que l’on pourra afficher sous forme d’un diagramme de Gantt. Cependant, il serait dommage de prendre une séquence au hasard parmi l’ensemble de séquences qui ne répondent qu’à un critère : elles sont valides. En effet, pour commencer, on peut vouloir minimiser le temps de réponse moyen des tâches. Pour cela, on coche les tâches autres que la tâche oisive, et on choisit dans le menu *Action* d’optimiser le temps de réponse des tâches cochées. Il ne reste alors plus qu’une seule séquence donnée sur la figure VI-1-6.

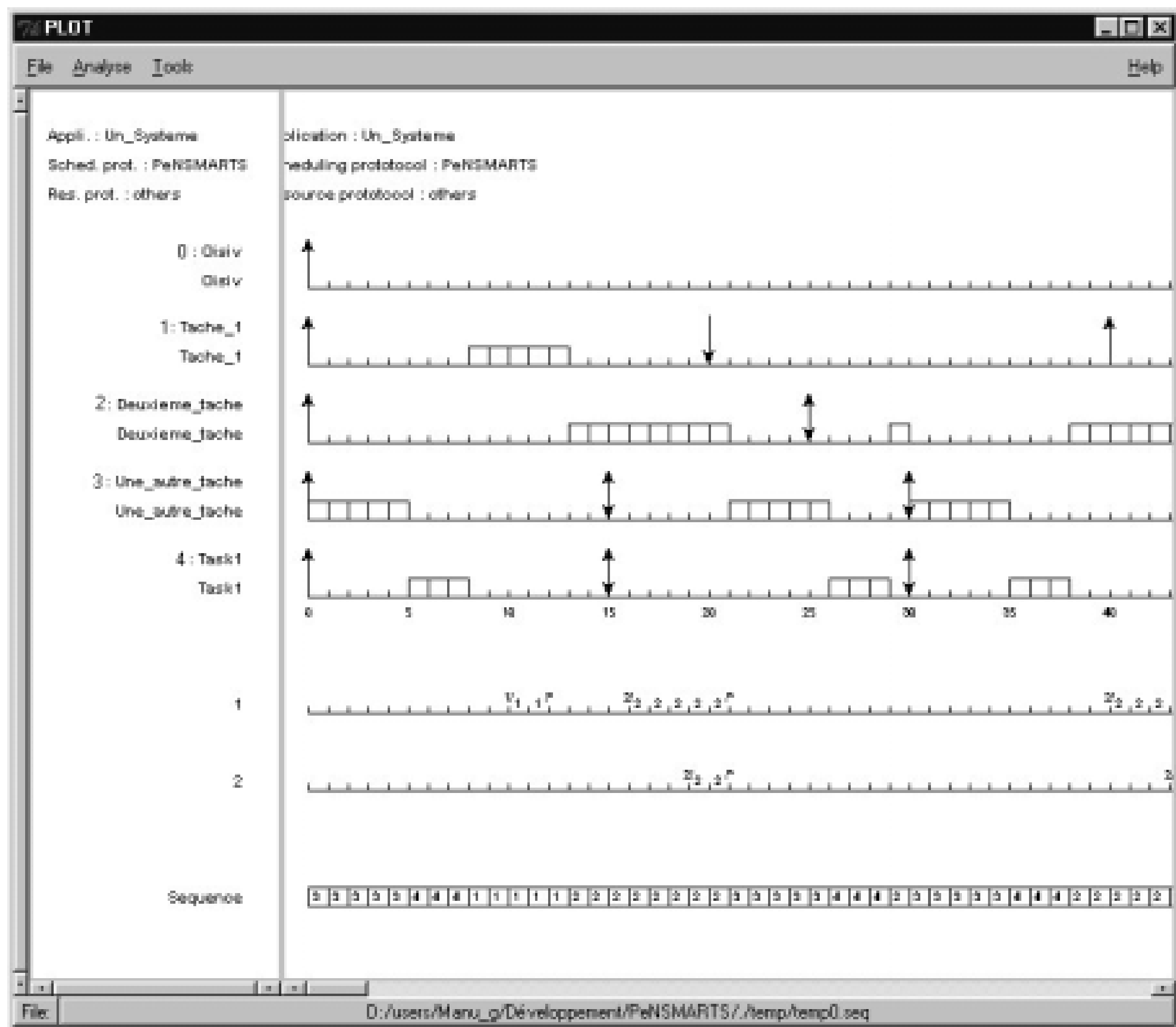


figure VI-1-6 : visualisation d'un diagramme de Gantt à l'aide de PLOT

Si cette séquence ne satisfait pas l'utilisateur, il peut annuler ce raffinement du *GM* afin par exemple, d'optimiser le temps de réponse maximal de la tâche Tache_1, qui à son goût, est très importante. Cela donne encore un grand nombre de séquences, représentées à l'aide de *3270* nœuds. Il reste alors, afin de minimiser le temps de réponse des autres tâches, à leur appliquer le critère minimisation du temps de réponse moyen, ce qui laisse *24* séquences.

Enfin, l'utilisateur peut imposer une borne sur le temps de réponse de la Tache_1, si par exemple c'est une tâche d'échantillonnage. Cela peut se faire via l'édition de contraintes absolues. Par exemple, il est possible d'imposer que le temps de réponse *TempsDeRéponse*(*Tache_1*)≤*Charge*(*Tache_1*)+*2*.

Cependant, si l'utilisateur veut minimiser la gigue de Tache_1, il y a de fortes chances pour qu'il soit obligé de construire le *GM* sans l'utilisation de contraintes de successeurs. Par contre, afin de diminuer dès le départ la taille du *GM* construit, il peut imposer directement une contrainte sur le temps de réponse de Tache_1. Par exemple, si le temps de réponse de Tache_1 est contraint à *7*, *PeNSMARTS* propose un *GM* composé de *6915* nœuds. Toutes les séquences d'ordonnancement contenues dans ce graphe voient Tache_1 avoir un temps de réponse de *7*, donc une gigue nulle.

VI-2 Etude de cas

Afin d'illustrer l'utilisation de *PeNSMARTS* sur un cas concret, nous détaillons ici une étude de cas.

VI-2.1 Description du procédé

L'aspect sécurité est le point essentiel de la gestion d'une mine. Le système de gestion de la sécurité concerne essentiellement le contrôle de deux fluides :

- eau (évacuation à partir d'un puisard)
- air (niveau de méthane)

Il s'agit pour l'application développée de contrôler le niveau d'eau qui doit être maintenu entre deux niveaux. L'infiltration naturelle du site augmente perpétuellement ce niveau, et une pompe a été installée afin, lorsque le niveau haut est atteint, d'être actionnée pour vider la mine jusqu'à atteindre le niveau bas. Parallèlement à cela, la concentration de méthane doit être surveillée à l'aide d'un capteur, afin d'éviter un « coup de grisou » : si un seuil d'alerte est dépassé l'alarme doit être déclenchée afin d'évacuer le personnel de la mine. Si un second seuil d'alerte est dépassé, la pompe ne doit pas être en fonctionnement, car la moindre étincelle pourrait provoquer une explosion.

Le pilotage de cet ensemble comprend donc, comme c'est illustré sur la figure VI-2-1, les éléments suivants :

- Une pompe à eau
- Deux capteurs tout ou rien de niveau d'eau: le capteur niveau bas (LLS pour *Low Level Sensor*), et le capteur niveau haut (HLS pour *High Level Sensor*)
- Un capteur de méthane MS (pour *Methan Sensor*)
- Une interface vers l'opérateur pour affichage de l'alarme

La mine doit fonctionner tant que la sécurité maximale peut être maintenue. Ainsi les indications générales de fonctionnement sont les suivantes :

- Une alarme doit être lancée sur la console de l'opérateur dès que la concentration de méthane dépasse le seuil *MS_L1* afin d'évacuer le personnel de la mine,
- La pompe doit être mise en route si le capteur HLS est immergé,
- La pompe doit être désactivée si le capteur LLS est à sec,
- La pompe ne doit pas fonctionner lorsque la concentration de méthane dépasse le seuil *MS_L2* afin d'éviter les riques d'explosion.

Avec *MS_L1* et *MS_L2* deux constantes prédéfinies.

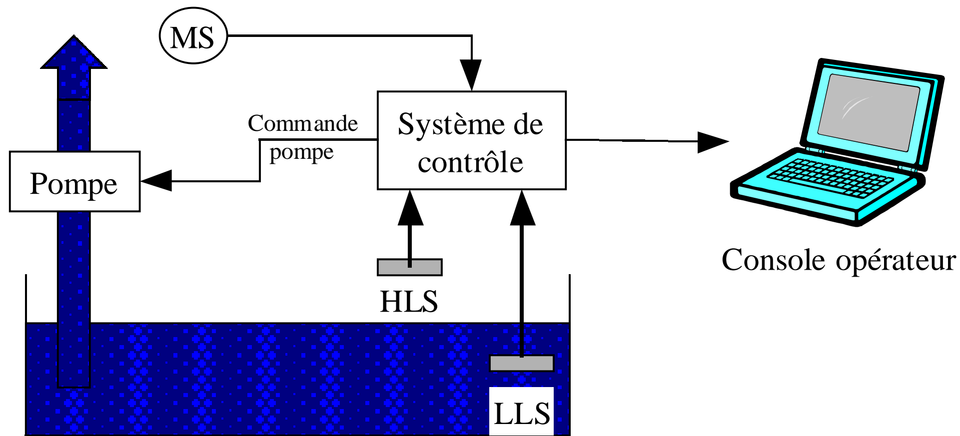


figure VI-2-1 : fonctionnement de la mine

VI-2.2 Spécification du contrôle

Le contrôle à effectuer sur le procédé peut donc être spécifié en SA-RT [Gol 93] par le diagramme de contexte présenté sur la figure VI-2-2.

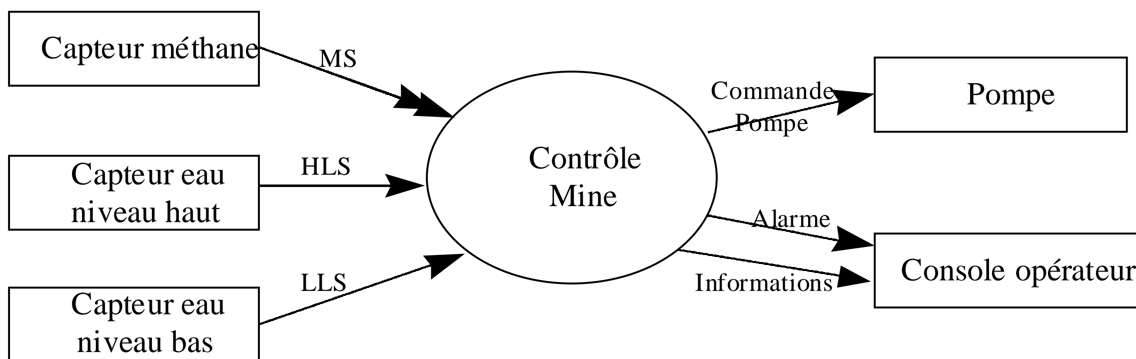


figure VI-2-2 : diagramme de contexte du contrôle de la mine

Les rectangles y représentent les bords, c'est à dire les points d'interaction entre le procédé et le contrôle. Par exemple, le *Capteur eau niveau haut* permet au contrôle d'obtenir des informations sur l'état du capteur *HLS*, et la pompe permet, via sa commande, d'agir sur le procédé. Les bords du modèle sont donc des capteurs (2 discrets HLS et LLS, et un capteur à valeurs continues MS, portant une information consultable en continu) et des actionneurs (la pompe que l'on peut allumer ou éteindre, et la console sur laquelle on peut activer et désactiver une alarme, et afficher en continu les informations sur le système).

Le diagramme préliminaire *SA-RT* présenté sur la figure VI-2-3 est donné à titre indicatif : il est destiné à donner au lecteur une idée du découpage de l'application en tâches et de ce à quoi elles sont dédiées. On y retrouve les informations et actions sur les bords. Le découpage des tâches en deux tâches d'acquisition permet au contrôle d'obtenir des informations sur le procédé et renseigne deux variables partagées *Niveau de méthane* et *Niveau d'eau* sur l'état du procédé. Le contrôle commande trois tâches de commande, l'une permettant d'actionner la pompe, l'autre l'alarme, et la dernière affichant les informations courantes sur le terminal de contrôle.

Il est important de constater que les deux variables utilisées pour afficher les informations sur le procédé sont partagées, et seront donc protégées à l'aide d'un sémaphore.

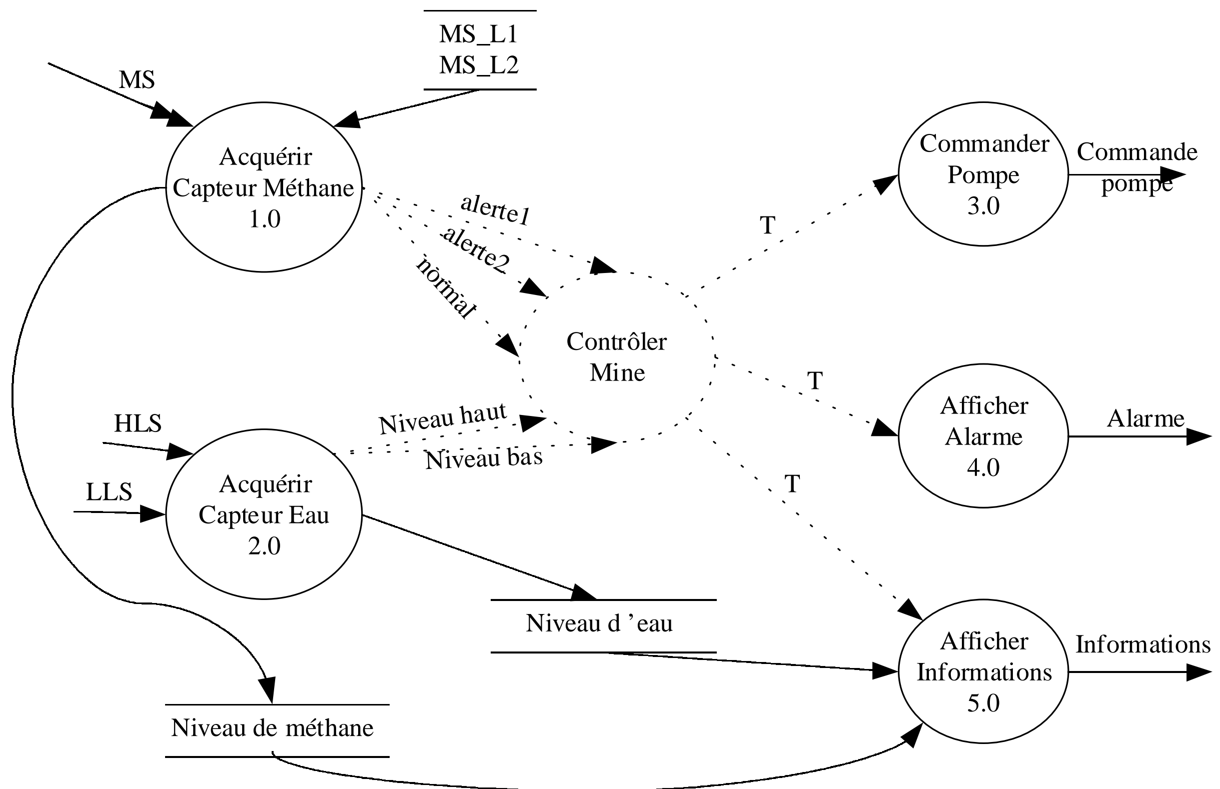


figure VI-2-3 : diagramme préliminaire du contrôle de la mine

VI-2.3 Implémentation

Le système est donc implémenté à l'aide de six tâches, et de trois sémaphores, permettant d'éviter tout accès concurrent aux variables partagées et au terminal. Les informations passées par les capteurs au contrôleur le sont par boîtes aux lettres, ainsi que les informations passées aux actionneurs. Par contre, les niveaux d'eau et de méthane sont stockés dans des modules de données, car la tâche *Afficher Informations* est très lente relativement aux tâches d'acquisition.

L'implémentation des tâches permet, après étude sur l'architecture cible, d'obtenir une abstraction composée uniquement de blocs de durée indépendantes et de primitives de communication et d'accès à des ressources. Celle-ci est donnée ci-dessous. Une unité de temps équivaut à un quantum de temps processeur, c'est à dire que son horloge lui permet de préempter une tâche chaque unité de temps.

```
Tâche Acquérir_Capteur_Méthane est
r1 = 0;
D1 = 100;
P1 = 100;
Début
      b1,1 = 8 ;
      -- Accès au capteur de méthane, calcul du niveau, et
      -- durée de l'accès à la boîte aux lettres ci-dessous
      Déposer_BAL(Méthane,donnée);
```

```
      b1,2 = 1 ;
      -- Durée de la prise du sémaphore
      Prendre_Sémaphore(Niveau_De_Méthane);
      b1,3 = 1 ;
      -- Durée de la modification de la variable et
      -- de la libération du sémaphore
      Rendre_Sémaphore(Niveau_De_Méthane);
Fin

Tâche Acquérir_Capteur_Eau est
r2 = 0;
D2 = 100;
P2 = 100;
Début
      b2,1 = 10 ;
      -- Accès aux deux capteurs d'eau, et
      -- durée de l'accès à la boîte aux lettres ci-dessous
      Déposer_BAL(Eau,donnée);
      b2,2 = 1 ;
      -- Durée de la prise du sémaphore
      Prendre_Sémaphore(Niveau_D_Eau);
      b2,3 = 1 ;
      -- Durée de la modification de la variable et
      -- de la libération du sémaphore
      Rendre_Sémaphore(Niveau_D_Eau);
Fin

Tâche Commander_Pompe est
r3 = 0;
D3 = 100;
P3 = 100;
Début
      b3,1 = 1 ;
      -- Accès à la boîte aux lettres ci-dessous
      Retirer_BAL(Moteur,donnée);
      b3,2 = 11 ;
      -- Accès au moteur
Fin

Tâche Afficher_Alarme est
r4 = 0;
D4 = 100;
P4 = 100;
Début
      b4,1 = 1 ;
      -- Accès à la boîte aux lettres ci-dessous
      Retirer_BAL(Alarme,donnée);
      b4,2 = 1;
      -- Accès au sémaphore protégeant l'écran
      Prendre_Sémaphore(Terminal);
      b4,3 = 23;
      -- Durée de l'affichage ou de l'extinction du signal d'alarme,
      -- Ainsi que de la libération du sémaphore
      Rendre_Sémaphore(Terminal);
Fin

Tâche Afficher_Informations est
r5 = 0;
D5 = 500;
P5 = 500;
Début
```

```
      b5,1 = 1 ;
      -- Durée de la prise du sémaphore
      Prendre_Sémaphore(Niveau_D_Eau);
      b5,2 = 1 ;
      -- Durée de la lecture et
      -- de la libération du sémaphore
      Rendre_Sémaphore(Niveau_D_Eau);
      b5,3 = 1 ;
      -- Durée de la prise du sémaphore
      Prendre_Sémaphore(Niveau_De_Méthane);
      b5,4 = 1 ;
      -- Durée de la lecture et
      -- de la libération du sémaphore
      Rendre_Sémaphore(Niveau_De_Méthane);
      b5,5 = 16 ;
      -- Durée des calcul de la prise du sémaphore
      Prendre_Sémaphore(Terminal);
      b5,6 = 50 ;
      -- Durée de l'accès au terminal et
      -- de la libération du sémaphore
      Rendre_Sémaphore(Terminal);
Fin

Tâche Contrôle est
r6 = 0;
D6 = 100;
P6 = 100;
Début
      b6,1 = 1 ;
      -- Durée de l'accès à la boîte aux lettres ci-dessous
      Retirer_BAL(Eau,donnée);
      b6,2 = 1 ;
      -- Durée de l'accès à la boîte aux lettres ci-dessous
      Retirer_BAL(Méthane,donnée);
      b6,3 = 10 ;
      -- Décision et accès à la boîte aux lettres ci-dessous
      Déposer_BAL(Moteur,donnée);
      b6,4 = 3 ;
      -- Décision et accès à la boîte aux lettres ci-dessous
      Déposer_BAL(Alarme,donnée);
Fin
```

On peut noter d'une part, que les périodes des tâches communicantes sont identiques. Cela conduit généralement à obtenir un *ppcm* des périodes assez faible. D'autre part, les durées des tâches et leurs actions sont évaluées dans le pire cas, par exemple, l'abstraction temporelle considère que des actions pour afficher ou éteindre l'alarme et lancer ou arrêter le moteur sont effectuées chaque fois, alors que cela est relativement rare. Cela conduit à une réflexion sur la prise en compte de conditionnelles au niveau de l'étude temporelle [CGC 97].

Après saisie de ce système de tâches à l'aide de *QTSD* ou à la main, l'outil *PeNSMARTS* permet l'obtention de quelques milliards de milliards de séquences d'ordonnancement valides avec l'utilisation des contraintes de successeur, stockées dans un *GM* de *26261* nœuds.

La minimisation du temps de réponse moyen laisse une seule séquence d'ordonnancement.

Notons, de plus, que ce système n'est ordonnançable par aucun algorithme au plus tôt (ce qui est le cas des algorithmes en-ligne les plus répandus). Pour illustrer ce fait, la figure VI-2-4 présente le début d'une séquence d'ordonnancement qu'aurait pu calculer un ordonnanceur *ED*, quel que soit le protocole de gestion de ressources utilisé.

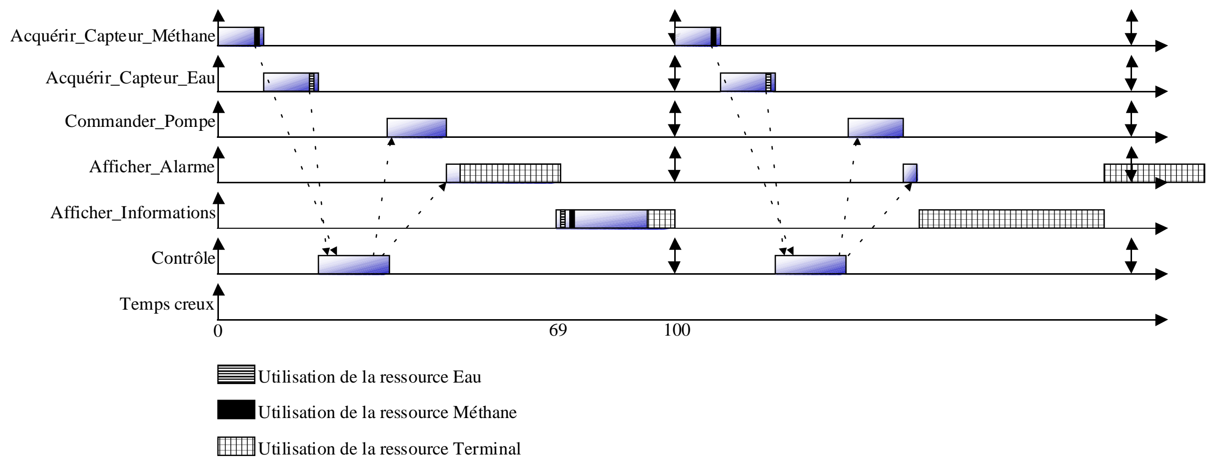


figure VI-2-4 : préfixe d'un ordonnancement au plus tôt

En effet, en utilisant un ordonnancement au plus tôt, la très longue section critique de la tâche *Afficher_Informations* débute avant la réactivation des autres tâches, ce qui empêche la tâche *Afficher_Alarme* de respecter sa seconde échéance. Toute séquence valide retarde le début de la section critique de *Afficher_Informations*, et n'est donc pas au plus tôt.

Ce genre de phénomène arrive assez souvent lorsque l'on utilise un terminal : l'application doit afficher des informations, ce qui est très coûteux en temps, alors que certaines tâches d'alarmes doivent pouvoir accéder au terminal de façon urgente.

CONCLUSION

Nous avons vu dans une première partie qu'il existait des algorithmes d'ordonnancement en-ligne polynomiaux optimaux permettant une validation temporelle analytique des systèmes de tâches ordonnancées (i.e. sans simulation). Lorsque les tâches partagent des ressources critiques, le problème de l'ordonnancement devient NP-difficile, à cause du problème de non préemptibilité des tâches en section critique par d'autres tâches en attente de leur(s) ressource(s). Plusieurs protocoles de gestion de ressources ont été mis en œuvre, visant à éradiquer les inversions de priorité. Cependant, ce n'est pas toujours suffisant pour pouvoir ordonnancer les systèmes de tâches très contraints : en effet, les phénomènes de blocage dus à l'attente d'une ressource ne peuvent être évités. La validation des algorithmes d'ordonnancement dans de tels cas est donc d'autant plus difficile qu'il y a de ressources critiques en jeu : en effet, dans ce cas, les conditions suffisantes d'ordonnançabilité se basent sur les temps de blocage maximaux qui peuvent être infligés aux tâches, ce qui augmente de façon artificielle la charge des systèmes de tâches lors de la validation. De plus, même la simulation, dont la durée nécessaire n'était pas connue lorsque des tâches sont différées, ne permet pas de valider de tels algorithmes d'ordonnancement sans intégrer toutes les durées de blocage dans les tâches puisque les algorithmes d'ordonnancement ne sont pas réguliers lorsque des ressources critiques sont en jeu (i.e. la pire durée n'est pas le pire cas).

L'approche hors-ligne est donc justifiée lorsque les systèmes de tâches à ordonnancer sont très contraints en termes de ressources critiques. Cependant, jusqu'ici les approches hors-ligne se sont cantonnées à l'étude des systèmes de tâches périodiques simultanées via leur décomposition en instances non périodiques. De plus, les systèmes de tâches qu'elles considèrent n'intègrent ni les ressources en lecture-écriture, proposées notamment par Ada 95, ni la possibilité de mixer des ressources critiques et des parties non préemptibles de façon simple. Pour finir, les approches hors-ligne proposent des recherches de séquences d'ordonnancement basées sur un seul critère : le retard maximal.

Notre étude a donc consisté en deux points :

- D'une part étudier le cas des systèmes de tâches périodiques différées. Cela nous a conduit à définir la notion de temps creux acycliques. Nous avons montré que toute séquence d'ordonnancement produite en plaçant des temps creux acycliques uniquement lorsqu'il n'y avait rien d'autre à faire était périodique de période $P=ppcm_{i=1..n}(P_i)$ après le dernier temps creux acyclique, qui se trouvait obligatoirement avant la date $r+P$ où r est la date de réveil la plus tardive. Nous avons vu que l'application de ce résultat permettait d'étendre toute étude hors-ligne aux systèmes de tâches différées, et de borner la durée de simulation nécessaire à l'étude de tels systèmes de façon quasi-universelle.

- D'autre part, nous avons étudié une technique d'énumération basée sur les réseaux de Petri dont le langage est l'ensemble de tous les ordonnancements possibles des tâches modélisées. La notion de temps y est introduite non pas explicitement par une temporisation (au sens temporel ou temporisé des RdP) mais implicitement par un fonctionnement sous la règle de tir maximal. Tous les ordonnancements sont donc obtenus par construction du graphe des marquages du RdP. Son intérêt est double : c'est d'une part un outil d'énumération assez simple à mettre en œuvre, et d'autre part, il permet de prendre en compte des primitives temps réel rarement proposées dans les autres approches, comme l'accès à des ressources en lecture ou en écriture, et la prise en compte simplifiée de parties non préemptives de tâches. Le modèle ayant la puissance d'une machine de Turing, toute structure peut être prise en compte par cet outil d'énumération. L'évolutivité aisée du modèle a été montrée lorsque nous avons montré comment passer simplement de l'hypothèse monoprocesseur à l'hypothèse multipro-

cesseur. Cette approche étant exponentielle en temps et en espace, une grande partie du travail de recherche consiste à stopper au plus tôt la construction de certaines branches du graphe des marquages menant à des séquences d'ordonnancement peu intéressantes. Nous utilisons à cette fin des techniques permettant de limiter le nombre de changements de contexte des tâches à l'aide de contraintes de successeur sur les transitions : cela a pour effet de supprimer des maillages du graphe pour n'en conserver que les arêtes externes. Nous utilisons aussi des contraintes sur les séquences que l'on veut obtenir qui seront résolues le plus tôt possible lors de la construction. Enfin, la forme en diamant du graphe des marquages permet d'extraire simplement, en un temps linéaire par rapport à la taille du graphe, des séquences ayant certaines propriétés qualitatives (optimalité sur différents critères basés sur le temps de réponse des tâches) ou quantitatives.

L'une des perspectives les plus prometteuses à l'utilisation d'une telle approche semble être une modélisation plus fine des tâches conditionnelles : en effet, lors de l'étude des systèmes de tâches, une abstraction basée sur ses caractéristiques temporelles et sur ses interactions avec les autres tâches est faite. Lorsqu'une tâche est composée de conditionnelles, ce qui est très souvent le cas (« si il se passe tel événement, alors j'opère tel traitement, sinon, je fais tel autre traitement »), les branches des conditionnelles sont réunies en une seule branche de durée la durée maximale des branches conditionnelles, et faisant les mêmes accès aux ressources que si les deux branches étaient exécutées en même temps. Cela entraîne parfois des cas pathologiques, et souvent des pires cas éloignés de la réalité. Il serait donc intéressant de prendre en compte au niveau du modèle réseau de Petri ces conditionnelles : une séquence d'ordonnancement étant alors un arbre, chaque embranchement correspondant à un test conditionnel.

BIBLIOGRAPHIE

[ABRW 91] N.C. Audsley, A. Burns, M.F. Richardson, A.J. Wellings, « *Hard real-time scheduling: the deadline monotonic approach* », Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta, pp. 127-132, 1991.

[ABRW 92] N.C. Audsley, A. Burns, M.F. Richardson, A.J. Wellings, « *Deadline monotonic scheduling theory* », Real-time Programming, pp. 55-60, 23-26 juin 1992.

[ABDTW 95] N.C. Audsley, A. Burns, R.I. Davis, K.W. Tindell,A.J. Wellings, « *Fixed priority pre-emptive scheduling: an historical perspective* », Real-Time Systems, 8, pp. 173-198; 1995.

[Ada 95] International Standard ISO/IEC 8652:1995 (E), « *Ada 95 Reference Manual* ».

[ATB 93] N.C. Audsley, K.W. Tindell, A. Burns, « *The end of the line for static cyclic scheduling ?* », proc. of the 5th Euromicro Workshop on Real-Time Systems, pp. 36-41, June 1993.

[Aud 91] N.C. Audsley, « *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times* », Technical Report YCS-164, University of York, 31 p., nov. 1991.

[Bab 96] J.P. Babau, « *Etude du comportement temporel des applications temps réel à contraintes strictes basées sur une analyse d'ordonnançabilité* », thèse de doctorat d'informatique, LISI/ENSMA, Futuroscope, 227 pp., 1996.

[Bak 91] T.P. Baker, « *Stack-based scheduling of real-time processes* », the Journal of Real-Time Systems, 3, pp. 67-99, 1991.

[Bar 96] J. Barnes, « *Programming in Ada95* », Addison-Wesley, Wokingham, England, 702 pp., 1996.

[BC 98] J.P. Babau, F. Cottet, « *Méthodologie d'analyse temporelle des applications temps réel à contraintes strictes* », APII-JESA, vol. 32, n°5-6, pp. 581-608, Hermes, Sept. 1998.

[BD 91] B. Berthomieu, M. Diaz, « *Modeling and verification of time dependent systems using time Petri nets* », IEEE Transactions on Software Engineering, 17(3), pp. 259-273, Mar 1991.

[BD 98] J.P. Beauvais, A.M. Déplanche, « *Affectation de tâches dans un système temps réel réparti* », Technique et Science Informatiques, 17(1), Jan 1998.

[Bea 96] J-P. Beauvais, « *Etude d'algorithmes de placement de tâches temps réel périodiques complexes dans un système réparti* », Thèse de Doctorat en Automatique et Informatique Appliquée, Ecole Centrale de Nantes, 182 p., 1996.

[Ber 98a] G. Berry, « *The foundations of Esterel* », to appear in Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte, editors, MIT Press, 1998.

[Ber 98b] G. Berry, « *The Esterel Primer* », Current version 2.0 for Esterel v5.10, 140 pp., 17 march 1998.

[BF 86] E. Best, C. Fernandez, « *Notations and terminology on Petri net theory* », Arbeitpapiere der Gesellschaft für Mathematik und Datenverarbeitung MBH 195, 29 p., Jan 1986.

[BFR 73] P. Bratley, M. Florian, P. Robillard, « *On scheduling with earliest start and due date constraints with applications to computing bounds for the (n/m/G/Fmax) problem* », Naval Research Logistic Quarterly, 20, pp. 57-67, Mar. 1973.

[BFR 75] P. Bratley, M. Florian, P. Robillard, « *Scheduling with earliest start and due date constraints on multiple machines* », Naval Research Logistic Quarterly, 22(1), pp. 165-173, 1975.

[BG 92] G. Berry, G. Gonthier, « T*he Esterel synchronous programming language: design, semantics, implementation* », Sience of Computer Programming, vol. 19, n.2, 1992, pp. 83-152.

[Bla 76a] J. Blazewicz, « *Scheduling dependant tasks with different arrival times to meet deadlines* », in E. Gelenben H. Beilner (eds), *Modeling and performance evaluation of computer systems*, Amsterdam, North-Holland, pp. 57-65, 1976.

[Bla 76b] J. Blazewicz, « *Deadline scheduling of tasks - a survey* », Foundations of Control Engineering, 1(4), pp. 203-216, 1976.

[BM 82] B. Berthomieu, M. Menasche, « *An enumerative approach for analyzing Time Petri nets* », 3rd European Workshop on Applications and Theory of Petri Nets, Varenna, Italy, Sept 1982.

[BMR 90] S.K. Baruah, A.K. Mok, L.E. Rosier, « *Preemptively scheduling hard-real-time sporadic tasks on one processor* », Proc. 11th IEEE Real-Time Systems Symposium, pp. 182-190, Dec 1990.

[BR 90] I.I. Bestuzheva, V.V. Rudnev, « *Timed Petri nets: classification and comparative analysis* », Automation and Remote Control, 51(10), pp. 1303-1318, 1990.

[BHR 90] S.K. Baruah, R.R. Howell, L.E. Rosier, « *Algorithms and complexity concernig the preemptive scheduling of periodic, real-time tasks on one processor* », Real-Time Systems, 2, pp. 301-324, 1990.

[Bry 92] R.E. Bryant, « *Symbolic boolean manipulation with ordered binary decision diagrams* », ACM Computing Surveys, Vol. 24, No. 3, pp. 293-318, sept. 1992.

[BS 74] K.R. Baker, Z.S. Su, « *Sequencing with due-dates and early start times to minimize maximum tardiness* », Naval Research Logistic Quarterly, 21, pp. 171-176, 1974.

[But 97] G.C. Buttazzo, « *Hard real-time computing systems: predictable scheduling algorithms and applications* », Kluwer Academic Publishers, 381 p., Sept. 1997.

[CC 88] J. Carlier, P. Chrétienne, « *Timed Petri nets schedules* », Advances in Petri nets, in Lecture Notes in Computer Science, 340, pp. 62-84, Springer Verlag,1988.

[CCG 84] J. Carlier, P. Chrétienne, C. Girault, « *Modeling scheduling problems with timed Petri nets* », Advances in Petri nets, in Lecture Notes in Computer Science, 188, pp. 62-82, Springer Verlag, 1984.

[CCH 98] F. Cottet, M. Courtes, M. Holle, « *Traitement de la gigue temporelle pour les ordonnancements temps réel par échéance* », actes de Real-Time Systems, Paris, 14-16 janvier 1998.

[CGC 96] A. Choquet-Geniet, D. Geniet, F. Cottet, « *Exhaustive computation of the scheduled task execution sequences of a real-time application* », Lecture Notes in Computer Science, 1135, 1996.

[CGC 97] A. Choquet-Geniet, D. Geniet, F. Cottet, « *Ordonnançabilité des sytèmes temps-réel conditionnels à contraintes strictes* », Proc. Real-Time Systems and Embedded Systems, pp. 295-309, ed. Teknéa, Paris.

[Chr 83] P. Chrétienne, « *Les réseaux de Petri temporisés* », Thèse d'Etat, Université Paris VI, Juin 1983.

[CV 93] A. Choquet-Geniet, G. Vidal-Naquet, , « *Réseaux de Petri et systèmes parallèles* », Ed. Armand Colin, 1993.

[CGL 93] E. Clarke, O. Grumberg, D. Long, « *Verification tools for finite-state concurrent systems* », LNCS 803, pp. 124-175, june 1993.

[CL 90] M. Chen, K. Lin, « *Dynamic priority ceilings: a concurrency protocol for real-time systems* », the Journal of Real-Time Systems, 2(4), pp. 325-346, 1990.

[CNRS 88] Groupe de Réflexion Temps Réel du CNRS, « *Le temps réel* », Technique et Science Informatiques, 7(5), pp. 493-500, 1988.

[CMM 67] R.W. Conway, W.L. Maxwell, L.W. Miller, « *Theory of scheduling* », Addison-Wesley, 1967.

[CSB 90] H. Chetto, M. Silly, T. Bouchentouf, « *Dynamic scheduling of real-time systems under precedence constraints* », the Journal of Real-Time Systems, vol. 2, pp. 181-194, 1990.

[Cot 97] F. Cottet, « *Traitement des signaux et acquisition des données* », Dunod, 352 p., 1997.

[Cot 99] F. Cottet, « *Etude de l'application temps réel embarquée: mission Pathfinder* », actes de l'école d'été temps réel, ETR'99, ENSMA, Futuroscope, pp. 109-117, 13-16 sept 1999.

[Dav 98] L. David, « *Gigue temporelle et ordonnancement par échéance dans les applications temps réel* », rapport de DEA, LISI-ENSMA, 49 p., 1998.

[DE 95] J. Desel, J. Esparza, « *Free choice Petri nets* », Cambridge University Press, 244 p., 1995.

[Der 74] M. Dertouzos, « *Control robotics: the procedural control of physical processors* », Proc. IFIP Congress, pp. 807-813, 1974.

[DM 89] M.L. Dertouzos, A.K. Mok, « *Multiprocessor on-line scheduling of hard real-time tasks* », IEEE Transactions on Software Engineering, 15(12), pp. 1497-1506, Dec 1989.

[DS 72] M.I. Dessouky, C.R. Margenthaler, « *The one-machine sequencing problem with early starts and due dates* », AIIE Transactions, 4(3), Sept. 1972.

[Duv 98] D. Duverney, « *Théorie des nombres : cours et exercices corrigés* », chap. 7, pp. 90, Dunod, 246 p., 1998.

[Ell 97] J.P. Elloy, « *Systèmes réactifs, synchronisme, tâches et événements* », actes de l'école d'été temps réel, ETR'97, 22-26 sept, ENSMA, Futuroscope, pp. 28-37, 1997.

[For 56] L.R. Ford Jr., « *Network flow theory* », The Rand Corporation, 293p., 1956.

[Gal 97] L. Gallon, « *Le modèle réseaux de Petri temporisés stochastiques : extensions et applications* », PhD. thesis, Université Paul Sabatier, Toulouse, dec. 1997.

[Gen 86] H.J. Genrich, « *Predicate/Transitions nets* », Advanced Course on Petri Nets, 57 p., Bad Honnef, 8-19 sept. 1986.

[GBGM 91] P. Le Guernic, M. Le Borgne, T. Gauthier, C. Le Maire, « *Programming real-time applications with Signal* », Another look at real-time programming, proc. of the IEEE, special issue, Sept. 1991.

[GJ 83] M.R. Garey, D.S. Johnson, « *Computers and intractability : A guide to the theory of NP-completeness* », 2nd Edition, W.H. Freeman, New York, 1983.

[GL 81] H.J. Genrich, K. Lautenbach, « *System modelling with high-level Petri nets* », Theoretical Computer Science, 13, pp. 109-136, 1981.

[GLLR 79] R.L. Graham, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, « *Optimization and approximation in deterministic sequencing and scheduling: A Survey* », Ann. Discrete Mathematics, 5, pp. 287-326, 1979.

[GM 95] M. Gondran, M. Minoux, « *Graphes et algorithmes* », 3ème édition, 588 p., Eyrolles, 1995.

[Gol 93] S. Goldsmith, « *A practical guide to real-time systems development* », Prentice Hall, 503 p., 1993.

[Gro 96] E. Grolleau, « *Ordonnancement de systèmes de tâches temps réel à l'aide de réseaux de Petri* », rapport de stage de DEA, LISI-ENSMA-Université de Paris-Sud, 64 p., 1996.

[Har 87] P.K. Harter, « *Response time in level structured systems* », ACM Transactions on Computer Systems, 5(3), pp. 232-248, 1987.

[HCRP 91] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, « *The synchronous dataflow programming language Lustre* », Proceedings of the IEEE, vol. 79, nr. 9. Sept. 1991.

[HLR 92] N. Halbwachs, F. Lagnier, C. Ratel, « *Programming and verifying critical systems by means of the synchronous data-flow programming language Lustre* », IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems, September 1992.

[HLR 96] J.-F. Hermant, L. Leboucher, N. Rivierre, « *Real-time fixed and dynamic priority driven scheduling algorithms: theory and experience* », Rapport de Recherche INRIA n°3081, 141 p., déc. 1996.

[HM 95] C. Hanen, A. Munier, « *Cyclic scheduling on parallel processors : an overview* », Chapt. 9, pp. 193-226, Scheduling theory and its applications, P. Chrétienne et al., éditions John Wiley & Sons, New York, 365 pp., 1995.

[HSSM 68] A. Holt, H. Saint, R. Shapiro, S. Marshall, « *Final report of the information system theory project* », Tech. Rep. RADC-TR-68-305, Rome Air Development Center, Griffis Air Force Base, New York, 352 p., Sept 1968.

[Jac 55] J.R. Jackson, « *Scheduling a production line to minimize maximum tardiness* », Research Report UCLA, Management Sciences Research Project, 1955.

[Jen 81] K. Jensen, « *Coloured Petri nets and the invariant-method* », Theoretical Computer Science, 14, pp. 317-336, 1981.

[Jen 94] K. Jensen, « *An introduction to the theoretical aspects of coloured Petri nets* », A Decade of Concurrency, in Lecture Notes in Computer Science, 803, pp. 230-272, 1994.

[Jen 96] K. Jensen, « *Coulored Petri nets, basic concepts, analysis methos and practical use* », Vol. 1, 2nd edition, Monographs in theoretical Computer Science, 234 p., Springer Verlag, 1996.

[Jen 98] K. Jensen, « *An introduction to the practical use of coloured Petri nets* », Lectures on Petri Nets: Applications, in Lecture Notes in Computer Science, 1492, pp. 237-292, 1998.

[JLKD 86] R. Janicki, P.E. Lauer, M. Koutny, R. Devillers, « *Concurrent and maximally concurrent evolution of nonsequential systems* », Theoretical Computer Science, 43(2-3), pp. 213-238, 1986.

[Jos 85] M. Joseph, « *On a problem in real-time computing* », Information Processing Letters, 20(4), pp. 173-177, 1985.

[JP 86] M. Joseph, P. Pandya, « *Finding response times in a real-time system* », Computer Journal,, 29(5), pp.390-395, 1986.

[JSM 91] K. Jeffay, D.F. Stanat, C.U. Martel, « *On non-preemptive scheduling of periodic and sporadic tasks* », Proc. IEEE Real-Time Systems Symposium, San Antonio, Texas, pp. 129-139, Dec 4-6 1991.

[Jua 99] G. Juanole, « *Modélisation et évaluation du protocole MAC du réseau CAN* », actes de ETR'99, pp. 187-200, Poitiers-Futuroscope, 13-16 sept. 1999.

[Kai 82] C. Kaiser, « *Exclusion mutuelle et ordonnancement par priorité* », Technique et Science Informatiques, 1982.

[Kar 72] R.M. Karp, « *Reducibility among combinatorial problems* », in *Complexity of Computer Computations*, pp. 85-103, R.E. Miller and J.W. Watcher editors, Plenum Press, New York, 1972.

[KN 80] K.H. Kim, M. Naghibdadeh, « *Prevention of task overruns in real-time non-preemptive multiprogramming systems* », Proc. of Perf., pp. 267-276, 1980.

[Knu 81] D.E. Knuth, « *Seminumerical algorithms* », in *The Art of Computer Programming*, vol. 2, 2nde édition, Addison Wesley, 689 p., 1981.

[KR 90] R.M. Karp, V. Ramachandran, « *Parallel algorithms for shared-memory machines* », pp. 869-935, chap. 17, Handbook of Theoritical Computer Science vol. A: Algorithms and Compexity, Ed. J. van Leeuwen, Elsevier Science Publishers, 1990.

[KRPOH 94] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M.G. Harbour, « *A practitioner's handbook for real-time analysis: guide to rate monotonic analysis for real-time systems* », 712 p., Kluwer Academic Publishers, 1994.

[Lab 74] J. Labetoulle, « *Un algorithme optimal pour la gestion des processus en temps réel* », Revue Française d'Automatique, Informatique et Recherche Opérationnelle, pp. 11-17, Fevr 1974.

[Leh 90] J.P. Lehoczky, « *Fixed priority scheduling of periodic task sets with arbitrary deadlines* », Proc. IEEE Real-Time Systems Symposium, pp. 201-209, 1990.

[LL 73] C.L. Liu, J.W. Layland, « *Scheduling algorithms for multiprogramming in a hard real-time envirronment* », Journal of the ACM, vol. 20, n°1, pp. 46-61, 1973.

[LM 80] J.Y.-T. Leung, M.L. Merrill, « *A note on preemptive scheduling of periodic real-time tasks* », Information Processing Letters, 11(3), pp. 115-118, Nov, 1980.

[LM 81] E.L. Lawler, C.U. Martel, « *Scheduling periodically occuring tasks on multiple processors* », Information Processing Letters, 12, pp. 9-12, Feb 1981.

[LRKB 77] J.K. Lenstra, A.H.G. Rinnooy Kan, P. Brucker, « *Complexity of machine scheduling problems* », Ann. Discrete Math., 1, pp. 343-362, Jan 1977.

[LSD 89] J.P. Lehoczky, L. Sha, Y. Ding, « *The rate monotonic scheduling algorithm: exact characterization and average case behavior* », Proc. 10th Real-Time Systems Symposium, pp. 166-171, Dec 1989.

[LSST 91] J.P. Lehoczky, L. Sha, J.K. Strosnider, H. Tokuda, « *Fixed priority scheduling theory for hard real-time systems* », in *Foundations of Real-Time Computing: Scheduling and resource management*, Kluwer Academic Publishers, pp. 1-30, 1991.

[LW 82] J.Y.-T. Leung, J. Whitehead, « *On the complexity of fixed-priority scheduling of periodic real-time tasks* », Performance Evaluation, 2, pp. 237-250, 1982.

[MA 98] Y. Manabe, S. Aoyagi, « *A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling* », Real-Time Systems, 14(2), pp. 171-181, Mars 1998.

[Mar 82] C.U. Martel, « *Preemptive scheduling with release times, deadlines and due times* », ACM Computer Journal, 29, pp. 812-829, Jul 1982.

[Mat 96] J. Mathai et al., « *Real-time systems, specification, verification and analysis* », Prentice Hall, International Series in Computer Science, 296p., 1996.

[McN 59] R. McNaughton, « *Scheduling with deadline and loss functions* », Management Sci., 12(1), pp. 1-12, 1959.

[MF 76] P.M. Merlin, D.J. Farber, « *Recoverability of communication protocols - Implications of a theoretical study* », IEEE Transactions on Communications, pp. 1036-1043, Sept 1976

[MB 83] M. Menasche, B. Berthomieu, « *Time Petri nets for analyzing and verifying time dependent communication protocols* », Protocol Specification, Testing, and Verification, III, H. Rudin and C.H. West (ed.), IFIP, pp. 161-172, 1983.

[MD 78] A.K. Mok, M.L. Dertouzos, « *Multiprocessor scheduling in a hard real-time environment* », Proc. 7th Texas Conference on Computer Systems, pp. 5.1-5.12, Nov 1978.

[Mill 93] K.L. McMillan, « *Symbolic model checking* », Kluwer academic publishers, Boston, 216 pp., 1993.

[Min 67] M. Minsky, « *Computation : Finite and infinite machines* », Englewood Cliffs, New Jersey: Prentice-Hall, 317 p., 1967.

[Mok 83] A.K. Mok, « *Fundamental design problems for the hard real-time environments* », Ph. D. MIT, May 1983.

[Mur 89] T. Murata, « *Petri nets: properties, analysis and applications* », Invited Paper, Proc. Of the IEEE, 17(4), pp. 541-580, Apr 1989.

[Pap 94] C.H. Papadimitriou, « *Computational complexity* », Addison-Wesley, 500 p., 1994.

[Pet 80] J.L. Peterson, « *Availability of some early english-language reports on Petri nets* », Newsletter of the Special Interest Group on Petri Nets and Related System Models n°4, pp. 21-27, Feb 1980.

[Pet 81] J.L. Peterson, « *Petri nets theory and the modeling of systems* », Prentice-Hall, Englewood Cliffs, New Jersey, pp. 290, Apr 1981

[Pet 62] C.A. Petri, « *Kommunikation mit automaten* », Bonn Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962, Second Edition, New York: Griffiss Air Force Base, Technical Report RADC-TR-65--377, Vol.1, Suppl. 1, English translation, 1966.

[PN 98] P. Puschner, R. Nossal, « *Testing the results of static worst-case execution-time analysis* », proc. IEEE Real-Time Systems Symposium, pp. 134-143, Madrid, Spain, dec. 2-4, 1998.

[PK 89] P. Pusher, C. Koza, « *Calculating the maximum execution time of real-time programs* », Real-Time Systems, vol. 1, n°2, pp. 159-176, 1989.

[PS 91] C.Y. Park, A.C. Shaw, « *Experiment with a program timing tool based on source-level timing schema* », IEEE Computer, vol. 24, n° 5, pp. 48-57, 1991.

[Ram 74] C. Ramchandani, « *Analysis of asynchronous concurrent systems by timed Petri nets* », Ph. D. Thesis, MIT, Cambridge, Project MAC Report MAC-TR-120, Feb. 1974.

[Ram 90] K. Ramamritham, « *Allocation and scheduling of complex periodic tasks* », Proc. 10th Int. Conf. On Distributed Computing Systems, Paris, France, pp. 108-115, Jun 1990.

[RC 99] P. Richard, F. Cottet, « *Ordonnancement temps réel monoprocesseur avec contraintes de précédence simple et généralisée* », Rapport de Recherche LISI n°9902, 1999.

[RCM 96] I. Ripoll, A. Crespo, A.K. Mok, « *Improvement in feasibility testing for real-time tasks* », the Journal of Real-Time Systems, 11(1), pp. 19-39, 1996.

[Ric 97] P. Richard, « *Contribution des réseaux de Petri à l'étude de problèmes de recherche opérationnelle* », thèse de doctorat d'informatique de l'Université François Rabelais de Tours, 173 pp., 1997.

[RTE 96] « *Real-time executive for multiprocessor systems, RTEMS, C Applications* », U.S. army missile command, Redstone Arsenal, Alabama 35898-5254, 254 pp.

[Saa 98] S. Saad-Bouzefrane, « *Etude temporelle des applications temps réel distribuées à contraintes strictes basée sur une analyse d'ordonnançabilité* », thèse de doctorat d'informatique, LISI/ENSMA, Futuroscope, 180 pp., 1998.

[SC 99] S. Saad-Bouzefrane, F. Cottet, « *Méthodologie d'analyse temporelle des applications temps réel réparties* », APII-JESA, vol. 33, n°3, pp. 251-284, Hermes, avril 1999.

[Ser 72] O. Serlin, « *Scheduling of time critical processes* », proc. Spring Joint Computers Conference, pp. 925-932, 1972.

[Shaw 89] A.C. Shaw, « *Reasoning about time in higher-level language software* », IEEE Transactions on Software engineering, vol. 15, n°7, pp. 875-889, 1989.

[SRL 87] L. Sha, R. Rajkumar, J.P. Lehoczky, « *Priority inheritance protocols : an approach to real-time synchronization* », Tech. Rep. CMU-CS-87-181, Carnegie-Mellon, 23 p., Dec 1987.

[SRL 90] L. Sha, R. Rajkumar, J.P. Lehoczky, « *Priority inheritance protocols : an approach to real-time synchronization* », IEEE Transactions on Computers, 39(9), pp. 1175-1185, Sept 1990.

[SS 93] L. Sha, S.S. Sathaye, « *A systematic approach to designing distributed real-time systems* », IEEE Computer, 26, pp. 68-78, sept 1993.

[SS 94] M. Spuri, J.A. Stankovic, « *How to integrate precedence constraints and shared resources in real-time scheduling* », IEEE Transactions on Computers, 43(12), pp. 1407-1412, 1994.

[SSNB 95] J.A. Stankovic, M. Spuri, M. Di Natale, G. Buttazzo, « *Implications of classical results for real-time systems* », IEEE Computer, 28(6), pp. 1-25, 1995.

[SSRB 98] J.A. Stankovic, M. Spuri, K. Ramamritham, G.C. Buttazzo, « Deadline scheduling for real-time systems: EDF and related algorithms », Kluwer Academic Publishers, Oct. 1998.

[Sta 88] J.A. Stankovic, « *Misconception about real–time computing* », IEEE Computer Magazine, n°21, vol. 10, pp. 0–19, 1988.

[Sta 90] P. Starke, « *Some properties of timed nets under the earliest firing rule* », Advances in Petri nets 1989, Venice, Jun 1988, in Lecture Notes in Computer Science, 424, pp. 418-432, 1990.

[Tan 99] A. Tanenbaum, « *Systèmes d'exploitation : systèmes centralisés, systèmes distribués* », 2ème édition, Dunod, 824 pp., 1999.

[TE 97] J.P. Thomesse, J.P. Elloy, « *Ecole d'été temps réel : applications, réseaux et systèmes* », actes de l'école d'été temps réel, ETR'97, ENSMA, Futuroscope, pp. 2-3, 22-26 sept 1997.

[VV-N 81] R. Valk, G. Vidal-Naquet, « *Petri nets and regular languages* », Journal of Computer and System Sciences, 23(3), pp. 399-325, Dec 1981.

[Wel 97] B.B. Welch, « *Pratical programming in Tcl and Tk* », 2nd Edition, Prentice Hall, 650 p., 1997.

[XP 90] J. Xu, D. Parnas, « *Scheduling processes with release times, deadlines, precedence, and exclusion relations* », IEEE Transactions on Software Engineering, 16(3), pp. 360-369, Mar 1990.

[XP 92] J. Xu, D. Parnas, « *Pre-run-time scheduling of processes with exclusion relations on nested or overlapping critical sections* », Phoenix Conference on Computers and Communications, pp. 6.4.7.1-6.4.7.9, Apr 1992.

[XP 93] J. Xu, D. Parnas, « *On satisfying timing constraints in hard real-time systems* », IEEE Transactions on Computers, 19(1), pp. 70-84, Jan 1993.

[Xu 93] J. Xu, « *Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations* », IEEE Transactions on Software Engineering, 19(2), pp. 139-153, Feb 1993.

[ZRS 87a] W. Zhao, K. Ramamritham, J.A. Stankovic, *« Scheduling tasks with resource requirements in hard real-time systems »*, IEEE Transactions on Software Engineering, SE-13, pp. 564-577, may, 1987.

[ZRS 87b] W. Zhao, K. Ramamritham, J.A. Stankovic, « *Preemptive scheduling under time and resource constraints* », IEEE Transactions on Computer, C-36(8), pp. 949-960, aug, 1987.

[ZS 94] Q. Zheng, K.G. Shin, « *On the ability of establishing real-time channels in point-to-point packet-switched networks* », IEEE Transactions on Communications, 42(2,3,4), 1994.

[Zub 91] W.M. Zubereck, « *Timed Petri nets definitions, properties, and applications* », Invited paper, Microelectron. Reliab., 31(4), pp. 627-644, 1991.

BIBLIOGRAPHIE LIÉE À L'ÉTUDE

Publications dans les actes de conférence d'audience nationale avec comité de lecture

E. Grolleau, A. Choquet-Geniet, F. Cottet, « *Ordonnancement optimal de systèmes de tâches temps réel à l'aide de réseaux de Petri* », actes de conférence AGIS'97, Automatique Génie informatique Image Signal, 9-11/12/97, Angers, pp. 239-246.

E. Grolleau, A. Choquet-Geniet, F. Cottet « *Cyclicité des ordonnancements au plus tôt des systèmes de tâches temps réel* », RenPar'10, Rencontres Francophones du Parallélisme, 9-12/06/98, Strasbourg, pp. 39-42.

E. Grolleau, A. Choquet-Geniet, F. Cottet, « *Validation de systèmes temps réel à l'aide de réseaux de Petri : PeNSMARTS* », AFADL'98, Approches Formelles dans l'Assistance au Développement de Logiciels, 30/10-1/11 1998, Poitiers-Futuroscope, pp. 137-148.

E. Grolleau, A. Choquet-Geniet, F. Cottet, « *Modélisation des systèmes temps réel par réseaux de Petri autonomes en vue de leur analyse hors-ligne* », MSR'99, Modélisation des Systèmes Réactifs, 24-25 mars 1999, Cachan.

Rapports de recherche

E. Grolleau, « Bases de Tcl/Tk », RR. 97002, LISI, ENSMA, Futuroscope, 1997, 52 p.

E. Grolleau, A. Choquet-Geniet, F. Cottet, « Ordonnancement optimal de systèmes de tâches temps réel à l'aide de réseaux de Petri », RR. 97004, LISI, ENSMA, Futuroscope, 1997, 8p.

E. Grolleau, A. Choquet-Geniet, F. Cottet, « Cyclicité d'ordonnancements au plus tôt des systèmes de tâches temps réel à contraintes strictes », RR. 97007, LISI, ENSMA, Futuroscope, 1997, 14p.

Communications avec résumé

E. Grolleau, « Projet PeNSMARTS, Petri Net Scheduling, Modeling & Analysis of Real-time Systems », session atelier d'ETR'97, Ecole d'été temps réel '97, p.235, Futuroscope, 22-26 sept. 1997.

E. Grolleau, « Validation de systèmes de tâches temps réel à l'aide de réseaux de Petri », Movep'98, MOdélisation et VErification des processus Parallèles, 6-9/07/98, Nantes, p.268.

E. Grolleau, « Projet PeNSMARTS, Petri Net Scheduling, Modeling & Analysis of Real-time Systems », session atelier d'ETR'99, Ecole d'été temps réel '99, p.233, Futuroscope, 13-16 sept. 1999.

ANNEXES

I-1 Etude du *ppcm*

Nous montrons ici que le *ppcm* d'un nombre arbitraire d'entiers n'est pas borné polynomialement. La preuve se base sur des résultats de complexité en ordonnancement temps réel.

Théorème I-1-1 : le *ppcm* de *n* entiers positifs arbitraires n'est pas borné polynomialement.

Preuve : La preuve est basée sur une synthèse de [LM 80][LW 82] et de [BHR 90]. Dans le premier article, Leung et Merrill ont montré qu'une méthode possible de test d'ordonnançabilité de systèmes de tâches était la simulation, et que la durée de simulation était bornée dans tous les cas par $r+2P$ avec $r=max(r_i)_{i=1..n}$ et $P=ppcm(P_i)_{i=1..n}$. Nous avons amélioré cette borne, montrant que la durée de simulation était au minimum P et au maximum $r+2P$, apportant un gain pouvant aller jusqu'à plus 50% de la durée. Cependant, la durée de simulation nécessaire à la vérification de la faisabilité d'un système est dans le cas général de l'ordre du *ppcm* des périodes. Un algorithme de test de faisabilité, appelons-le *simulation*, d'un système de tâches indépendantes consiste donc à tester la validité d'une séquence produite par *ED* pendant la durée de simulation. L'algorithme *simulation* a une complexité de l'ordre du *ppcm* des périodes.

Leung et Merrill ont d'abord défini le problème ***SCP***[13] et ont montré qu'une instance I_{SCP} d'un problème *SCP* pouvait être réduite polynomialement à un système de tâches indépendantes I_T tel que « I_T n'est pas ordonnançable » équivaut à « I_{SCP} a une solution ». Etant donné que *simulation* permet de dire si I_T est ordonnançable ou non, il permet de résoudre n'importe quelle instance de *SCP*. Dans [LW 82], il est montré par réduction du problème ***CLIQUE***[14] [GJ 83] au problème *SCP* que *SCP* est pseudo-polynomial complet (on dit aussi *NP*-complet-unaire ou encore *NP*-complet au sens faible).

Baruah et al [BHR 90] ont amélioré ce résultat en réduisant le **3-*SAT***[15] à *SCP*, montrant ainsi que *SCP* est *NP*-complet au sens fort.

Etant donné que *simulation* est un algorithme solvant *SCP* qui est donc *NP*-complet au sens fort, et que *simulation* est polynomial dans le *ppcm* des périodes de tâches, *ppcm* ne peut pas être borné pseudo-polynomialement.

❑

Cependant, il est montré dans [Duv 98] $ppcm(1,2,..,m)$ est borné par 3^m.

[13] *K-Simultaneous Congruences Problem* : soit un ensemble de couples $E=\{(a_1,b_1),\ldots,(a_n,b_n)\}$ d'entiers avec les $a_i\in \aleph$, $b_i\in \aleph^+$, et un entier $K\geq 2$. Existe-t-il un sous-ensemble $F\subseteq E$ avec $|F|\geq K$ tel que $\forall(a_i,b_i)\in F$, $\exists x\in \aleph^+$, $x\equiv a_i\ [b_i]$. En d'autres termes, un couple $c=(a,b)$ peut être vu comme une suite périodique de premier élément $c_0 = a$, et $c_1 = a+b,\ldots, c_m = a+mb$. Le problème est de trouver un sous-ensemble de cardinal au moins K de ces suites tel que toutes ont un élément commun.

[14] *CLIQUE* : Soit un graphe non orienté $G=(V,E)$, et un entier positif $K\leq|V|$. Y-a-t'il une *clique* (graphe complet) de taille K ou plus dans G ? Problème montré *NP*-complet au sens faible dans [Kar 72].

[15] 3-*SAT* : Problème directement issu de *SATISFIABILITY*, problème de base considéré *NP*-complet au sens fort.

I-2 Notions de complexité

Cette annexe précise les concepts de complexité [GJ 83][Pap 94].

Le problème de savoir si un problème est effectivement solvable par l'informatique se base principalement sur deux points :

- L'existence d'un algorithme résolvant ce problème,
- La complexité du problème, c'est à dire le temps et l'espace qui sont nécessaires à la recherche de la solution du problème posé.

Ce second point est primordial car nombre de problèmes classiques sont solvables algorithmiquement, ne serait-ce que par énumération de toutes les solutions possibles, mais le temps ou l'espace nécessaires à certains algorithmes rendent leur utilisation non réaliste.

Afin d'étudier plus avant ce problème, lié au temps et à l'espace nécessaires à l'exécution d'un algorithme, la communauté de l'étude de complexité a classé les problèmes en deux catégories, et a utilisé des modèles abstraits de la machine de Van Neumann, la plus connue étant la machine de Turing. Nous avons vu cependant en section II-2.1.2 que d'autres modèles d'expression équivalente comme les machines à registres pouvaient être utilisés.

Les deux catégories de problème étudiés sont d'une part les **problèmes de décision**, c'est à dire dont la solution est oui ou non, et d'autre part les **problèmes de recherche** dont la problématique est d'exhiber un ensemble de solutions.

Par exemple, un des problèmes de décision que nous avons abordé est « Le système de tâches *S* est-il ordonnançable ? ». Nous avons vu que dans certains cas particuliers comme $|1|r_i,C_i,D_i=P_i,P_i|$, on pouvait répondre à cette question par oui ou non en un temps $O(n)$ en utilisant la CNS d'ordonnançabilité par *ED* (qui est optimal dans ce contexte) $\sum_{i=1}^{n}\frac{C_i}{P_i} \leq 1$.

Un des problèmes de recherche auquel nous avons été confrontés est « Trouver une séquence d'ordonnancement valide d'un système », ce qui dans tous les cas de tâches périodiques est au moins exponentiel, puisqu'une solution (en l'occurrence une séquence d'ordonnancement) est de taille exponentielle (au minimum le *ppcm* des périodes) par rapport au système de tâches considéré.

Les problèmes de recherche englobent les problèmes de décision, puisqu'un problème de décision est un problème de recherche pour lequel l'espace des solutions est {oui, non}. Dans cette annexe, nous étudions la complexité des problèmes de décision puis nous donnons des éléments d'étude de la complexité des problèmes de recherche..

Les problèmes sont classifiés par catégories suivant l'existence, ou non, d'algorithmes fonctionnant en un temps acceptable.

On parle de temps acceptable lorsque le temps nécessaire à la décision d'une solution est facteur d'un polynôme de la taille de l'entrée (des données) du problème. Les problèmes solvables en un temps polynomial forment la classe ***P*** (pour polynomiaux).

On parle de temps non acceptable lorsque la durée du traitement nécessaire à la recherche d'une solution est exponentielle par rapport à la taille des données entrées. Généralement, les problèmes concernés sont étudiés par des techniques heuristiques ou bien à l'aide d'algorithmes probabilistes. Cependant, il est intéressant de classer les problèmes de complexité non acceptables par difficulté.

La première classe de problèmes non polynomiaux est la classe ***NP***. *NP* ne signifie pas non-polynomiaux mais polynomiaux non déterministe. Ce sont des problèmes que l'on pourrait résoudre en un temps polynomial si l'on disposait d'une machine de Turing non déterministe (on parle aussi de machine à oracle). Typiquement, une machine à oracle permet de

transformer un arbre de recherche en une seule branche : la bonne. En effet, dans les problèmes solvables par construction d'un arbre de recherche, la complexité est due au fait que l'algorithme peut faire de nombreux retours arrières et avoir à construire tout un arbre (en un temps exponentiel) pour trouver une solution qui est une feuille de l'arbre. Une machine à oracle permet de considérer qu'à partir d'un nœud, on sait à l'aide de l'oracle, quel est le bon nœud fils à construire.

On conjecture que $P \neq NP$, c'est à dire que les problèmes de la classe *NP* nécessitent au moins un temps exponentiel de traitement.

Les problèmes *NP* ne sont cependant pas, loin s'en faut, les problèmes les plus complexes que l'on peut trouver : il existe des problèmes solvables en temps polynomial à l'aide d'une machine à oracle utilisant une autre machine à oracle, on parle alors de la classe NP^{NP}, et ainsi de suite.

Les problèmes peuvent donc être classifiés de cette façon dans *P*, *NP*, NP^{NP} etc… Toute la difficulté de la complexité réside, lorsqu'un nouveau problème se pose, à le placer dans une classe de complexité. En effet, un problème de *P* est dans *NP* mais un problème pour lequel on a trouvé un algorithme exponentiel n'a-t'il pas une solution polynomiale ?

Afin de s'assurer de la difficulté d'un problème, un problème de référence, *SAT* pour *satisfiability*, a été choisi. Ce problème est considéré comme le problème le plus dur solvable à l'aide d'une machine à oracle. Il s'agit, étant donné un ensemble de formule logiques composées d'un ensemble de variables booléennes, de décider s'il existe au moins une affectation des variables telle que les formules soient vraies.

Ce problème est la référence *NP*, on dit qu'il est ***NP*-complet**. Tous les problèmes *NP* aussi durs que *SAT* sont *NP*-complets. Le terme aussi dur, en complexité, est basé sur la notion de **réduction polynomiale**. On dit qu'un problème p_1 se réduit polynomialement à un problème p_2 (on note alors $p_1 \leq_p p_2$) si il existe un algorithme polynomial permettant de transformer une entrée du problème p_1 en entrée du problème p_2 tel que la réponse de l'algorithme de p_2 soit oui si et seulement si la réponse au problème initial par l'algorithme solvant p_1 est oui. La classe des problèmes *NP* auxquels on peut réduire polynomialement *SAT* est appelée la classe *NP*-complet : ce sont les problèmes *NP* aussi difficiles que *SAT*.

Certains auteurs font la distinction entre les problèmes NP et leur complémentaire (i.e. si répondre oui à un problème est aussi dur que de répondre non à un problème *NP*). Un problème dont le complémentaire est dans *NP* est un problème **co-*NP***.

Il existe des problèmes auxquels on peut réduire polynomialement *SAT*, mais dont l'appartenance à NP n'a pas été montrée. Dans ce cas, on parle de problèmes ***NP*-difficiles**.

La figure I-2-1 schématise par des ensembles le (sous-ensemble) des classes algorithmiques présentées. L'inclusion d'une classe *A* dans une classe *B* signifie que les problèmes de *A* se réduisent polynomialement à un problème de *B*.

Enfin il existe une autre classe beaucoup utilisée en algorithmique mais rarement clairement définie : la classe des problèmes **pseudo-polynomiaux**. Les problèmes pseudo polynomiaux sont les problèmes concernés par le codage des nombres (ou lettres) dans une base différente que 1. En effet, supposons qu'un algorithme prenant un entier *n* en entrée nécessite *n* pas, chaque pas étant de complexité constante. L'entier *n* entré est vraisemblablement codé dans une base, prenons *2* puisque nous sommes en informatique. La taille de l'entrée est donc $log_2(n)$, et l'algorithme nécessite *n* pas, c'est à dire une exponentielle de la taille de l'entrée. Cependant, si *n* avait été donné en base 1 (c'est à dire sous la forme de *n* bâtons), le problème serait considéré comme linéaire. L'algorithme n'est pas polynomial, et il n'est pas forcément dans *NP*. On parle alors d'un algorithme pseudo-polynomial.

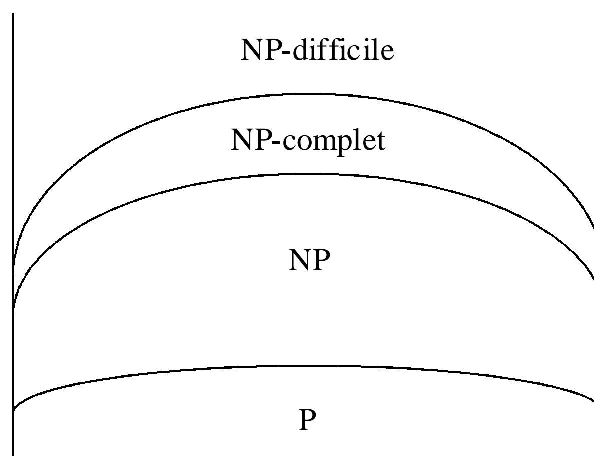


figure I-2-1 : hiérarchie simplifiée des classes de complexité algorithmique

I-3 Fonctionnement d'un réseau de Petri avec la règle de tir maximal

Nous fournissons dans cette annexe un algorithme de calcul des ensembles de transitions franchissables à partir d'un marquage pour un RdP avec la règle de tir maximal.

Il s'agit, à partir d'un marquage, d'obtenir tous les marquages accessibles par le tir d'un ensemble maximal (au sens de l'inclusion) de transitions franchissables.

Sur la figure I-3-1, les ensembles $\{t_1,t_3,t_4\}$, $\{t_2,t_3\}$, et $\{t_2, t_4\}$ sont franchissables.

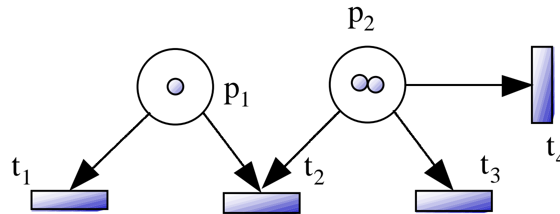


figure I-3-1 : un réseau de Petri avec la règle de tir maximal

L'algorithme utilisé dans *PeNSMARTS* se base sur la décomposition du tir en une phase d'enclenchement (consommation des jetons) de toutes les transitions possibles jusqu'à ce qu'aucune transition non encore enclenchée (pour éviter la réentrance) ne soit franchissable. Enfin le déclenchement de toutes les transitions enclenchées correspond à la production des jetons.

Pour commencer, nous définissons un ordre total sur les transitions du RdP avec pour t_i, $t_j \in T$, $t_i<t_j$ si et seulement si $i<j$.

L'algorithme procède par énumération d'ensembles de transitions franchissables simultanément. Le but est d'éviter les arrangements des ensembles de transitions, et de n'en obtenir que les combinaisons. L'algorithme utilise une pile de transitions enclenchées, dans laquelle les transitions enclenchées sont empilées dans l'ordre croissant, donc le sommet de la pile est la transition enclenchée ayant le plus grand indice. Nous procédons en deux phases : la phase d'empilement, qui correspond à la création d'un ensemble maximal de transitions franchissables simultanément, et la phase de dépilement, permettant de passer d'un ensemble à un autre.

```
Procédure Empilement (Rdp : ModèleRdP, M : Marquage,
                        P : Pile_de_transitions) est
-- P est la pile contenant les transitions déjà enclenchées
-- M est le marquage prenant en compte les jetons consommés par les
-- transitions enclenchées
-- RdP est le RdP composé notamment d'un ensemble de transitions T,
-- d'un ensemble de places Q, et des matrices de consommation W⁻ et de
-- production W⁺
      F: Ensemble de transitions;
      M': Marquage ← M;
Début
      F ← {tᵢ∈T / M'(tᵢ)≥W⁻(tᵢ)};
      -- F est l'ensemble des transitions franchissables pour M
      Si P≠∅ alors
            F ← F − {tᵢ∈F / tᵢ≤Sommet(P)};
            -- On retire de F les transitions dont l'indice n'est pas
            -- supérieur au sommet de la pile, évite les symétries et
```

```
            -- la réentrance
      Fin si;
      Tant que F≠∅ faire
      -- Tant que des transitions d'indice supérieur aux transitions
      -- enclenchées existent
            ti← Min(F);
            M' ← M' – W-(ti);
            -- On enclenche la transition franchissable suivant
            -- le sommet de P
            Empiler(P,ti);
            -- ti fait partie des transitions enclenchées, et c'est
            -- celle d'indice le plus grand parmi celles-ci
            F ← F – {tj∈F / M'(ti)<W-(ti)};
            -- On supprime de F les transitions qui étaient en conflit
            -- avec ti
      Fait
      F ← {ti∈T / M'(ti)≥W-(ti)}–{tj∈P};
      -- F est l'ensemble des transitions franchissables après
      -- enclenchement des transitions de P, desquelles nous enlevons
      -- les transitions déjà enclenchées (non réetrance)
      Si F=∅ alors
      -- L'ensemble est maximal
            ∀ti∈P, M'←M'+W+(ti);
            -- Les transitions enclenchées sont déclenchées
            CréerFils(M',P);
            -- M' est accessible par le tir des transitions de P
            -- avec la règle de tir maximal
      Fin si
      -- Si F≠∅, c'est que l'ensemble n'est pas maximal et qu'il a été
      -- traité avant par l'algorithme d'énumération
Fin
```

Voici la procédure de dépilement, permettant de passer d'un ensemble de transitions franchissables à un autre. Son principe est de dépiler le sommet de la pile en annulant son enclenchement, puis de trouver une autre transition franchissable, d'indice supérieur, qui devient alors le nouveau sommet de la pile, et qui permettra peut être via la procédure empilement de trouver un nouvel ensemble maximal de transitions franchissables.

```
Procédure Dépilement (Rdp : ModèleRdP, M : Marquage,
                      P : Pile_de_transitions) est
-- P est la pile contenant les transitions enclenchées dans
-- l'ensemble de transitions franchissables entièrement traité
-- précédemment.
-- M est le marquage prenant en compte les jetons consommés par les
-- transitions enclenchées
-- RdP est le RdP composé notamment d'un ensemble de transitions T,
-- d'un ensemble de places Q, et des matrices de consommation W- et de
-- production W+
      F: Ensemble de transitions;
Début
      Tant que P≠∅ faire
      -- Tant que tous les ensembles de transitions franchissables n'ont
      -- pas été énumérés
            tmin ← Sommet(P);
            -- tmin est la transition considérée comme sommet la fois
            -- d'avant
            M ← M + W-(tmin);
            -- l'enclenchement de tmin est annulé
```

```
            F ← {tᵢ∈T / M'(tᵢ)≥W⁻(tᵢ)};
            F ← F – {tᵢ∈F / tᵢ≤t_min};
            -- On ne s'intéresse pas aux transitions d'indice inférieur à
            -- t_min ni à t_min car elles ont déjà été traitées
            Si F≠∅ alors
            -- On peut commencer un nouvel ensemble ayant le pour
            -- début de pile P à partir de la plus petite transition
            -- franchissable
                  tᵢ ← Min(F);
                  Empiler(P,tᵢ);
                  M ← M - W⁻(tᵢ);
                  -- tᵢ est enclenchée
                  Empilement(RdP,M,P);
                  -- Le processus de construction d'un ensemble commence
            Fin si
            -- Si aucun nouvel ensemble ne peut être construit à partir
            -- de P, on continue à dépiler P
      Fait
Fin
```

La construction de tous les ensembles maximaux de transitions franchissables est initiée par l'appel `Dépilement(RdP, MarquageCourant, P0)` où P_0 est une pile ne contenant que la transition fictive t_0 avec $\forall t \in T$, $t_0 \leq t$.

Le principe de l'algorithme est illustré sur l'arbre de la figure I-3-2. Chaque nœud y représente la pile de transition enclenchées, chaque boucle de la procédure *Dépilement* s'y traduit par un retour d'un nœud vers le nœud père, alors que chaque appel et chaque boucle de la procédure *Empilement* se traduit par la construction d'un nœud fils. Enfin, les feuilles barrées sont refusées car ne correspondent pas à des ensembles de transitions maximaux, les autres sont des ensembles maximaux de transitions franchissables simultanément.

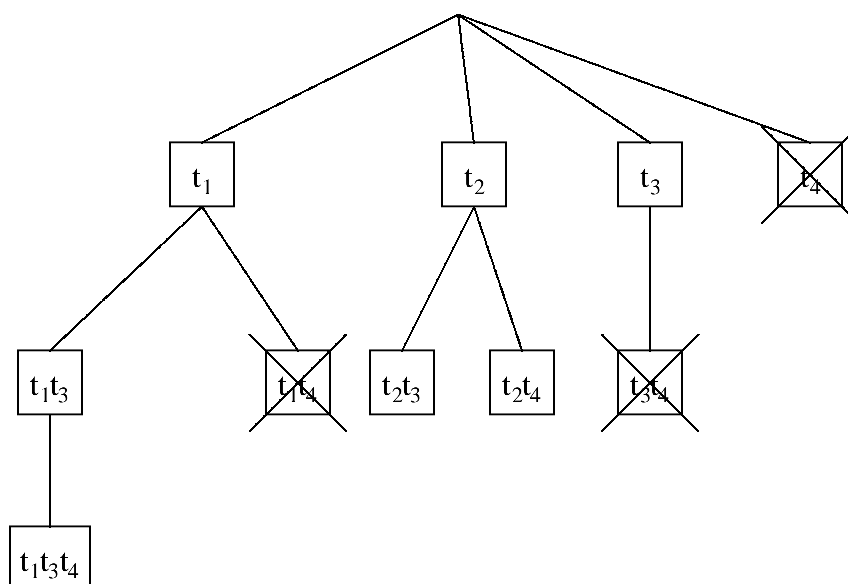


figure I-3-2 : fonctionnement de l'algorithme de construction des ensembles maximaux de transitions franchissables

I-4 Preuves relatives au résultat de cyclicité des ordonnancements

Lemme III-1-1 : Soit S un système de tâches ordonnancé dans le contexte $|1|r_i,C_i,D_i,P_i,prmptnoprmpt,prec,resRW|$, avec U_S=1. Le nombre de temps creux présents dans une séquence d'ordonnancement valide de S est borné.

Preuve : Soit un système de tâches S dont certaines sont différées, de charge U_S=1 (donc *100%*). Nous étudions le comportement temporel d'une tâche τ_i de S : notons $k_i=\left\lfloor \frac{r-r_i}{P_i} \right\rfloor$ (voir figure I-4-1) le nombre de requêtes entièrement traitées de τ_i dans l'intervalle $[0..r[$ où $r=max_{i=1..n}\{r_i\}$.

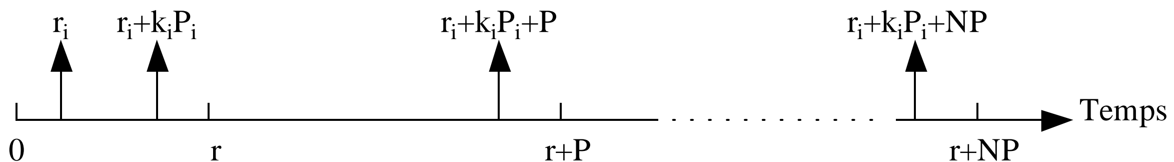


figure I-4-1 : activations de τ_i au cours du temps

Soit $N \in \mathbb{N}$ un entier arbitraire. Dans l'intervalle $[r_i+k_iP_i.. r_i+k_iP_i+NP[$, la tâche τ_i doit avoir terminé NP/P_i requêtes. La tâche τ_i nécessite donc $(k_i+NP/P_i)C_i$ unités de temps processeur dans l'intervalle $[0.. r_i+k_iP_i+NP[$, sous peine de faute temporelle.

Si l'on considère le système de tâches entier, le processeur devra traiter au moins $\sum_{i=1}^{n}\left(k_i+\frac{NP}{P_i}\right)C_i$ quanta de temps dans l'intervalle $max_{i=1..n}\{[0.. r_i+k_iP_i+NP[\}$. Cet intervalle est inclus ou égal à $[0..r+NP]$ étant donné que par définition $r_i+k_iP_i \le r$. Cela signifie que au moins $NPU_S+\sum_{i=1}^{n}k_iC_i$ quanta de temps correspondant au système de tâches entier doivent être traités dans l'intervalle $[0..r+NP[$.

Or par hypothèse, U_S=1, d'où au moins $NP+\sum_{i=1}^{n}k_iC_i$ quanta de temps sont à traiter par 1 processeur dans l'intervalle $[0..r+NP[$ pour N quelconque. Il ne peut donc y avoir plus de $r-\sum_{i=1}^{n}k_iC_i$ temps creux dans une séquence d'ordonnancement arbitrairement longue.

❑

Lemme III-1-4 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i|$ tel que U_S=1. La charge restant à traiter à un instant $t+P$ ne peut pas être inférieure à celle qui restait à traiter à la date t pour $t \ge r$. C'est à dire :

$\forall \Delta \ge 0,\ C_{rest}(r+P+\Delta) \ge C_{rest}(r+\Delta)$

Preuve : ce lemme permet de familiariser le lecteur avec l'évolution de la charge restante.

D'après la Définition III-1-2,

$$\sum_{i=1}^{P} C_{rest}(r+\Delta+i) = \sum_{i=1}^{P} C_{req}(r+\Delta+i) + \max\{0, C_{rest}(r+\Delta+i-1) - 1\} \quad \textbf{(a)}$$

D'où $\sum_{i=1}^{P} C_{rest}(r+\Delta+i) \geq \sum_{i=1}^{P} C_{req}(r+\Delta+i) + C_{rest}(r+\Delta+i-1) - 1$ **(b)**

Puisque la charge est de *100%*, d'après le Lemme III-1-3, on peut écrire

$$\sum_{i=1}^{P} C_{rest}(r+\Delta+i) \geq P + \sum_{i=1}^{P} C_{rest}(r+\Delta+i-1) - 1 \quad \textbf{(b)}$$

Ce qui implique immédiatement $\sum_{i=1}^{P} C_{rest}(r+\Delta+i) \geq \sum_{i=1}^{P} C_{rest}(r+\Delta+i-1)$ **(c)**

D'où le lemme.

❑

Proposition III-1-2 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i|$ tel que U_S=1. Toute séquence d'ordonnancement conservative de S valide sur l'intervalle $[0..t_c+P+1[$ laisse le système dans le même état à l'instant t_c+P+1 qu'à l'instant t_c+1.

Preuve : Par définition de t_c, $C_{rest}(t_c)=0$, donc $C_{rest}(t_c+1)$ est donnée uniquement par les requêtes des tâches activées à la date t_c+1. D'après la définition de t_c, il n'y a plus de temps creux après cette date, donc en particulier il n'y en a pas dans l'intervalle $[t_c+1..t_c+P+1]$ et le processeur traite donc exactement P unités de temps de charge. Or puisque la charge est de *100%*, il y a justement P unités de temps de charge à traiter dans $[t_c+1..t_c+P+1]$. Donc la charge à traiter à l'instant t_c+P+1 est donnée uniquement par les requêtes des tâches activées à la date t_c+P+1, qui, de par la périodicité des tâches, sont les mêmes qu'à la date t_c+1 si et seulement si toutes les tâches actives à la date t_c+P+1 l'étaient aussi à la date t_c+1. C'est le cas si et seulement si $\forall i \in \{1..n\}$, $r_i<t_c+P_i+1$. La preuve est obtenue par contradiction : supposons qu'il existe $j \in \{1..n\}$, tel que $r_j \geq t_c+P_j+1$.

En sommant la charge instantanée de 1 à P, on obtient :

$$\sum_{i=1}^{P} C_{rest}(t_c+i) = \sum_{i=1}^{P} C_{req}(t_c+i) + \max\{0, C_{rest}(t_c+i-1) - 1\} \quad \textbf{(a)}$$

Etant donné qu'il n'y a aucun temps creux à partir de la date t_c+1, et qu'il y a un temps creux à la date t_c, $C_{rest}(t_c)=0$ et $C_{rest}(t_c+i)>0$ $\forall i>0$. Donc $max\{0,C_{rest}(t_c)-1\}=0=1-1$, d'où :

$$\sum_{i=1}^{P} C_{rest}(t_c+i) = 1 + \sum_{i=1}^{P} C_{req}(t_c+i) + C_{rest}(t_c+i-1) - 1 \quad \textbf{(a)}$$

Dans l'intervalle $[t_c+1..t_c+P]$, chaque tâche τ_i est activée au plus P/P_i fois, mais par hypothèse, τ_j n'est activée que P/P_j-1 fois. La charge étant de *100%*, la somme des activations et réactivations dans cet intervalle nécessite au plus $P-C_j$ unités de temps. Donc (a) implique :

$$\sum_{i=1}^{P} C_{rest}(t_c+i) \leq 1 + P - C_j + \sum_{i=1}^{P} C_{rest}(t_c+i-1) - 1 \quad \textbf{(b)}$$

Ce qui s'écrit, en sortant le 1 de la somme :

$$\sum_{i=1}^{P} C_{rest}(t_c+i) \leq 1 - C_j + \sum_{i=1}^{P} C_{rest}(t_c+i-1) \quad \textbf{(b)}$$

D'où $C_{rest}(t_c+P) \leq 1-C_j+C_{rest}(t_c)$ **(b)**

Or par définition de t_c, $C_{rest}(t_c)=0$, d'où

$C_{rest}(t_c+P) \leq 1-C_j$ **(b)**

Etant donné que les tâches ne peuvent pas être de durée nulle, on a $C_j \geq 1$. D'où (b) donne $C_{rest}(t_c+P)=0$, ce qui contredit la définition de t_c.

Cela prouve que $\forall i \in \{1..n\}$, $r_i < t_c+P_i+1$, ainsi que la proposition.

❑

Lemme III-1-7 : Les attentes qui se trouvent après la date P englobent les attentes qui se trouvent une méta-période plus tôt. C'est à dire $\forall \Delta \geq 0$, $A(\Delta) \subseteq A(P+\Delta)$.

Preuve : La preuve est donnée par récurrence sur la construction de $A(t)$.

Pour commencer, nous montrons que $\forall \Delta \geq 0$, $A_0(\Delta) \subseteq A_0(P+\Delta)$.

Soit $\{\tau_i,k,\tau_j\} \in A_0(\Delta)$, par définition de A_0, $\Delta \geq r_i$, d'où $\Delta+P \geq r_i$.

De plus, par définition de A_0, $k+rb_{i,j}\left\lfloor \frac{\Delta - r_i}{P_i} \right\rfloor > db_{j,i}\left\lfloor \frac{P_j+\Delta - r_j}{P_j} \right\rfloor$ **(a)**

Or par hypothèse sur les tâches communicantes, $\frac{rb_{i,j}}{P_i} = \frac{db_{j,i}}{P_j}$ **(b)**

En ajoutant P fois (b) à (a), on obtient :

$$P\frac{rb_{i,j}}{P_i} + k + rb_{i,j}\left\lfloor \frac{\Delta - r_i}{P_i} \right\rfloor > P\frac{db_{j,i}}{P_j} + db_{j,i}\left\lfloor \frac{P_j+\Delta - r_j}{P_j} \right\rfloor \quad \textbf{(c)}$$

Or puisque $P=ppcm_{q=1..n}\{P_q\}$, $\frac{P}{P_i}$ et $\frac{P}{P_j}$ sont des entiers, on peut écrire :

$$k + rb_{i,j}\left\lfloor \frac{P+\Delta - r_i}{P_i} \right\rfloor > db_{j,i}\left\lfloor \frac{P+P_j+\Delta - r_j}{P_j} \right\rfloor \quad \textbf{(c)}$$

Donc $\{\tau_i,k,\tau_j\} \in A_0(P+\Delta)$, d'où $A_0(\Delta) \subseteq A_0(P+\Delta)$.

L'hypothèse de récurrence est que $\forall \Delta \geq 0$, $A_{m-1}(\Delta) \subseteq A_{m-1}(P+\Delta)$. Il nous reste à montrer que $\forall \Delta \geq 0$, $A_m(\Delta) \subseteq A_m(P+\Delta)$.

Soit $\{\tau_i,p,\tau_j\} \in A_m(\Delta)$, alors on a :

- soit $\{\tau_i,p,\tau_j\} \in A_{m-1}(\Delta)$, et par hypothèse de récurrence, $\{\tau_i,p,\tau_j\} \in A_{m-1}(P+\Delta)$,
- ou alors $\exists \{\tau_j,q,\tau_k\} \in A_{m-1}(\Delta)$, avec $p + rb_{i,j}\left\lfloor \frac{\Delta - r_i}{P_i} \right\rfloor \geq db_{j,i}\left| \frac{\Delta - r_j}{P_j} \right|$ et $CD_{j,i,r} > CR_{j,k,q}$. Ce qui implique, en ajoutant P fois (b), $p + P\frac{rb_{i,j}}{P_i} + rb_{i,j}\left\lfloor \frac{\Delta - r_i}{P_i} \right\rfloor \geq P\frac{db_{j,i}}{P_j} + db_{j,i}\left| \frac{\Delta - r_j}{P_j} \right|$,

  d'où $\{\tau_i,p,\tau_j\} \in A_m(P+\Delta)$, d'où $A_m(\Delta) \subseteq A_m(P+\Delta)$.

❑

Lemme III-1-11 : Soit S un système de tâches pris dans le contexte $\|1\|ri,C_i,D_i,P_i,prec,resRW\|$ tel que $U_S=1$. La charge restant à traiter à un instant $t+P$ ne peut pas être inférieure à celle qui restait à traiter à la date t pour $t \geq r$. C'est à dire : $\forall \Delta \geq 0$, $C_{rest}(r+P+\Delta) \geq C_{rest}(r+\Delta)$

Preuve : D'après la Définition III-1-5,

$$\sum_{i=1}^{P} C_{rest}(r+\Delta+i) = \sum_{i=1}^{P} C_{req}(r+\Delta+i) + C_{rest}(r+\Delta+i-1) - \gamma(r+\Delta+i-1) \quad \textbf{(a)}$$

Or par définition $\gamma(t)\leq 1$, d'où $\sum_{i=1}^{P} C_{rest}(r+\Delta+i) \geq \sum_{i=1}^{P} C_{req}(r+\Delta+i) + C_{rest}(r+\Delta+i-1) - 1$ **(b)**

Puisque la charge est de *100%*, d'après le Lemme III-1-10, on peut écrire

$$\sum_{i=1}^{P} C_{rest}(r+\Delta+i) \geq P + \sum_{i=1}^{P} C_{rest}(r+\Delta+i-1) - 1 \qquad \textbf{(b)}$$

La suite de la preuve est identique à celle du Lemme III-1-4.

❏

Lemme III-1-12 :Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i, prec,resRW|$ tel que $U_S=1$. Si il y a un temps creux à un instant $t\geq r$, alors il n'y en a pas à la date $t+P$. C'est à dire :

$\forall\Delta\geq 0$, $\gamma(r+\Delta)=0 \Rightarrow \gamma(r+P+\Delta)=1$

Preuve : D'après la Définition III-1-5,

$$\sum_{i=1}^{P} C_{rest}(r+\Delta+i) = \sum_{i=1}^{P} C_{req}(r+\Delta+i) + C_{rest}(r+\Delta+i-1) - \gamma(r+\Delta+i-1) \qquad \textbf{(a)}$$

Puisque la charge est de *100%*, d'après le Lemme III-1-10, on peut écrire

$$\sum_{i=1}^{P} C_{rest}(r+\Delta+i) = P + \sum_{i=1}^{P} C_{rest}(r+\Delta+i-1) - \gamma(r+\Delta+i-1) \qquad \textbf{(b)}$$

Or par hypothèse du lemme, $\gamma(r+\Delta)=0$, d'où

$$\sum_{i=1}^{P} C_{rest}(r+\Delta+i) = P + C_{rest}(r+\Delta) + \sum_{i=2}^{P} C_{rest}(r+\Delta+i-1) - \gamma(r+\Delta+i) \qquad \textbf{(c)}$$

Comme par définition $0\leq\gamma(t)\leq 1$, (c) devient

$$\sum_{i=1}^{P} C_{rest}(r+\Delta+i) \geq 1 + C_{rest}(r+\Delta) + \sum_{i=2}^{P} C_{rest}(r+\Delta+i-1) \qquad \textbf{(d)}$$

D'où $C_{rest}(r+P+\Delta)\geq 1+ C_{rest}(r+\Delta)$ **(e)**

Or par hypothèse du lemme, $\gamma(r+\Delta)=0$, ce qui équivaut par définition à

$C_{rest}(r+\Delta)= C_{att}(r+\Delta)$ **(f)**

(e) et (f) impliquent $C_{rest}(r+P+\Delta)\geq 1+ C_{att}(r+\Delta)$ **(g)**

Comme d'après le Lemme III-1-9, $C_{att}(r+\Delta)= C_{att}(r+P+\Delta)$ $\forall\Delta\geq 0$, (g) devient $C_{rest}(r+P+\Delta)\geq 1+ C_{att}(r+P+\Delta)$, ce qui, par définition, signifie que $\gamma(r+P+\Delta)=1$.

❏

Proposition III-1-5 : Soit S un système de tâches pris dans le contexte $|1|ri,C_i,D_i,P_i, prec,resRW|$ tel que $U_S=1$. Toute séquence d'ordonnancement conservative valide de S sur l'intervalle $[0..t_c+P+1[$ laisse le système dans le même état à l'instant t_c+P+1 qu'à t_c+1.

Preuve : Le même raisonnement que celui utilisé dans la Proposition III-1-2 nous permet d'arriver aux «équations suivantes :

1. $C_{rest}(t_c)=C_{att}(t_c)$ (d'après la définition des temps creux)
2. $C_{rest}(t_c+1)=C_{att}(t_c)+C_{req}(t_c+1)$ (la charge à l'instant suivant t_c est celle de t_c plus la charge amenée par les requêtes)
3. $C_{rest}(t_c+P)=C_{att}(t_c)+1$ (le processeur a traité exactement $P-1$ unités de temps de charge en P unités de temps car il y a eu exactement 1 temps creux sur l'intervalle $[t_c..t_c+P[$)

Etudions la $3^{ème}$ équation : nous savons d'après le Lemme III-1-7 que les attentes à l'instant t_c sont incluses ou égales aux attentes à l'instant t_c+P, d'où $C_{att}(t_c+P) \geq C_{att}(t_c)$. Etant donné que par définition de t_c, il n'y a pas de temps creux à l'instant t_c+P, le processeur peut traiter une unité de temps, donc $C_{att}(t_c+P)$ ne peut pas être supérieur à $C_{att}(t_c)$, sinon il y aurait un temps creux à la date t_c+P puisque le charge restante est donnée par $C_{att}(t_c)+1$. Le processeur peut donc traiter l'unité de temps en sus de $C_{att}(t_c)=C_{att}(t_c+P)$. Puisque les charges en attente sont identiques aux dates t_c et t_c+P nous déduisons que les chaînes d'attentes sont identiques. A la date t_c+P+1, la charge restante est donc donnée par $C_{att}(t_c+P)+C_{req}(t_c+P+1)$. Comme dans la preuve de la Proposition III-1-2, nous pouvons montrer que les requêtes à la date t_c+P+1 sont identiques aux requêtes à la date t_c+P+1. Il nous faut donc identifier si les tâches impliquées dans les attentes sont les mêmes à la date t_c+P+1 qu'à la date t_c+1 : c'est le cas d'après le périodicité des chaînes d'attente. Nous en déduisons le lemme.

❑

I-5 Algorithme de placement de tâches évitant les changements de contexte inutiles

```
Fonction Affecter (S : Séquence) retourne Séquence
-- Entrée : S est un tableau de P m-uplets, c'est à dire une séquence
-- fournie par le modèle
-- Retour : renvoie la même séquence S' avec placement, i.e.
-- la tâche de S'(t,i) est affectée au processeur i
S' : Séquence
Début
      Pour i de 1 à m faire
      -- m est le nombre de processeurs
      -- Les processeurs sont alloués de façon arbitraire
      -- pour commencer
            S'(1,i)←S(1,i)
      Fait
      Pour t de 2 à P faire
      -- Pour chaque m-uplet de la séquence
            Pour i de 1 à m faire
                  S'(t,i)←∅
            Fait
            Pour i de 1 à m faire
                  Si ∃j/S(t,i)=S(t-1,j) alors
                  -- La tâche n'est pas préemptée et ne doit pas l'être
                        Stock←S'(t,j)
                        -- On mémorise la tâche qui pourrait déjà être
                        -- sur le processeur j afin de la déplacer
                        S'(t,j)=S(t,i)
                        -- La tâche est affectée sur le même processeur que
                        -- précédemment
                        Si Stock≠∅ alors
                              Trouver k tel que S'(t,k)= ∅
                              -- On cherche le 1er processeur libre
                              S'(t,k)←Stock
                              -- Afin d'y affecter la tâche déplacée
                        FinSi
                  Sinon
                        Trouver k tel que S'(t,k)= ∅
                        -- On cherche le 1er processeur libre
                        S'(t,k)←S(t,i)
                  Finsi
            Fait
      Fait
      Retourner S'
Fin
```

Avec une complexité de $P \times m^3$, cet algorithme élimine les changements de contexte pouvant être évités.

INDEX

$\rightarrow$ ... *Voir* Précède
$\geq_{\chi}$ ... *Voir* Puissance d'ordonnançabilité
τ_0 ... *Voir* Tâche oisive
$\tau_{i,k}$ ... *Voir* $k^{ème}$ occurrence
3-*SAT* ... 209
A(t) ... Voir Attente
ABR ... *Voir* Arbre binaire de recherche
Absence de migration ... 165
Actionneur ... 6
AD(*t*) ... *Voir* Attentes directes
Algorithme au plus tôt. *Voir* Algorithme conservatif
Algorithme conservatif ... 29
Algorithme d'ordonnancement dynamique ... 29
Algorithme d'ordonnancement statique ... 29
Algorithme d'Audsley ... 37
Algorithme d'ordonnancement optimal ... 28
Algorithme en-ligne ... 29
Algorithme hors-ligne ... 29
Application temps réel ... 6
Arbre binaire de recherche ... 139
Attente ... 94
Attente directe ... 94
Attente indirecte ... 95
Backtracking ... Voir Retour arrière
BDD ... *Voir* Binary decision diagram
B_i ... *Voir* Temps de blocage maximal
Binary decision diagram ... 7
Blocage d'évitement ... 51
blocage direct ... 47
Blocage fatal ... 9
Blocage transitif ... 49
Boîte aux lettres ... 10
Bornitude ... 67
Branch and bound ... *Voir* Séparation et évaluation
Capteur ... 6
C_{att} ... *Voir* Charge en attente
Charge d'un système ... 18
Charge en attente ... 96
Charge instantanée ... *Voir* Charge restante
Charge restante ... 88
C_i ... *Voir* Charge processeur
Classe *NPVoir* Problèmes non déterministes polynomiaux
Classe *P* ... *Voir* Problèmes polyniomaux
CLI ... 16
CLIQUE ... 209
CNS ... *Voir* Condition nécessaire et suffisante
Communication
  Asynchrone ... 10
  Par boîte aux lettres ... *Voir* Asynchrone
  Par événement ... 12
  Par rendez-vous ... 12
  Par signal ... *Voir* Par événement
co-*NP* ... 211
Contexte d'étude d'ordonnançabilité ... 23
Contrainte absolue ... 147
Contrainte de précédence ... 21
Contrainte de successeur ... 146
Contraintes terminales ... 68
Contrôleur ... 6
C_{req} ... *Voir* Requête instantannée
C_{rest} ... *Voir* Charge restante *Voir* Charge restante
Critère arborescent ... 159
CS ... *Voir* Condition suffisante
$db_{i,b}$ ... *Voir* Nombre de dépôts dans une *BAL*
Deadline Driven Scheduling Voir Earliest Deadline
Deadline Monotonic ... 35
Deadlock ... *Voir* Blocage fatal
$début_{i,k}$ ... 19
Découpage en forme normale ... 43
D_i ... *Voir* Délai critique
$d_{i,k}$ ... *Voir* $k^{ème}$ échéance
Diagramme de charge processeur ... 86
Diamant ... 134
DM ... *Voir Deadline Monotonic*
Earliest Deadline ... 38
ED ... *Voir Earliest Deadline*
ESTEREL ... 7
Exclusion mutuelle ... 14
Exécutif temps réel ... 9
Exécution au plus tôt ... 150
F ... *Voir* Flot
Fiablement ordonnançable ... 28
Flot ... 65
Franchissement d'une transition ... 66
Gigue temporelle ... 20
GM ... *Voir* Graphe des marquages
Graphe de classe ... 72
Horloge temps réel ... 19
HTR ... Voir Horloge temps réel
Hyperpavé ... 136
Hypothèse asynchrone ... 6
Hypothèse synchrone ... 6
Inactif ... *Voir* Oisif
Instance ... *Voir* occurrence
Instant critique ... 33
Interblocage ... 47
Inverse Deadline ... *Voir Deadline Monotonic*
Jitter ... *Voir* Gigue temporelle
K-Simultaneous congruences ... 209
Langages réactifs ... 7
Langages synchrones ... 7
Latence ... 19
$latence_{i,k}$ ... *Voir* Latence
Laxité ... 20
Least Laxity ... 41
LL ... *Voir Least Laxity*
LUSTRE ... 7
M^0 ... *Voir* Absence de migration
M_0 ... *Voir* Marquage initial
M^1 ... *Voir* Migration faible
M^2 ... *Voir* Migration forte
Machine à registres ... 74
Machine de Turing ... 73
Marquage initial ... 66
Marquage stérile ... 144
Marquage terminal ... 68
Masquage des interruptions ... 16

Méta-période 31
Migration 165
Migration faible 165
Migration forte 165
Minimiser le retard 154
Minimiser le taux de réaction 153
Minimiser le temps de réponse 151
Model checking 7
Modélisation par RdP 125
Moniteur 15
Niveau de préemption 54
Noyau
  Attendre_Evt 13
  Cliquer_Evt 13
  Créer_BAL 11
  Créer_Evt 13
  Créer_Sémaphore 15
  Creer_Tâche 10
  Déposer_BAL 11
  Effacer_Evt 13
  Lancer_Tâche 10
  Prendre_Sémaphore 15
  Retirer_BAL 11
  Signaler_Evt 13
  Supprimer_Tâche 10
Noyau temps réel 9
NP-complet 211
NP-difficile 211
Objet protégé 15
Oisif 27
Ordonnançable 28
Ordonnancement 9
Ordonnancement régulier 30
PAP *Voir* Protocole d'allocation de la pile
P_i *Voir* Période
Place 65
Pondération du flot 65
PPH *Voir* Protocole à priorités héritées
PPP *Voir* Protocole à priorité plafond
PPPD *Voir* Protocole à priorité plafond dynamique
Préemption 8
Primitive temps réel 9
Problème de décision 210
Problème de recherche 210
Procédé 6
Protocole à priorité plafond 50
Protocole à priorité plafond dynamique 51
Protocole à priorités héritées 49
Protocole d'allocation de la pile 54
Pseudo-polynomial 211
PTR *Voir* Primitive temps réel
Puissance d'ordonnançabilité 28
Q *Voir* Place
Rate Monotonic 33
Rate Monotonic Analysis 35
$rb_{j,b}$ *Voir* Nombre de retraits d'une *BAL*
réduction polynomiale 211
Réentrance 19
Règle de tir maximal 78
Relative Urgency *Voir Earliest Deadline*
Rendez-vous 12
Requête instantanée 88
Réseau de Petri
  Centre d'un langage terminal 69
  Définition 66
  Etiquetage ε-libre 67
  Etiquetage à ε-transitions 67
  Etiquetage libre 67
  Etiqueté 67
  Flot 65
  Franchissement d'une transition 66
  Graphe d'accessibilité 67
  Graphe des marquages 67
  Langage 68
  Langage terminal 68
  M(ti>M' 66
  Marquage coloré 70
  Marquage courant 66
  Marquage initial 66
  Marquages accessibles 67
  Place 65
  Pondération du flot 65
  Réseau sous-jacent 66
  Transition 65
Réseau de Petri temporel 71
  Définition 71
  Intervalle de sensibilisation 71
Réseau places-transitions 65, 69
Réseau P-temporisé 75
Réseau T-temporisé 75
Ressource critique 14
Retard 154
Retour arrière 144
r_i *Voir* Date de réveil
$r_{i,k}$ *Voir* $k^{ème}$ activation
RM *Voir Rate monotonic*
RMA *See Rate Monotonic Analysis*
RTC 115
SA-RT 6
SCP Voir *K-Simultaneous congruences*
Section critique 14
Sémaphore 14
Sémaphore binaire 15
Séparation et évaluation 58
Séquence valide 27
Séquenceur 29
Serveur périodique de sporadique 22
SIGNAL 7
STI 16
Surcoût 30
Sur-échantillonnage 21
Système temps réel 6
T *Voir* Transition
Tâche 8
  τ_0 *Voir* Oisive
  τ_c *Voir* Creuse
  Active *Voir* Prête
  Apériodique 23
  Bloquée 9
  Charge processeur 18

Créée 9
Creuse..27
Cyclique 17
Date de première activation...............................18
Date de réveil ...*Voir* Date de première activation
Délai critique 18
Différée 18
Echéance 18
Echéance sur requête 19
Emettrice 11
En attente...9
Endormie 9
Exécutée 9
Existante 9
Indépendante 21
$k^{ème}$ activation 18
Occurrence 18
Offset ....*Voir* Date de première activation
Oisive 27
Paramètre C_i .... *Voir* Charge processeur
Paramètre D_i .... *Voir* Délai critique
Paramètre $d_{i,k}$ .... Voir $k^{ème}$ échéance
Paramètre P_i .... *Voir* Période
Paramètre r_i ......*Voir* Date de première activation
Paramètre $r_{i,k}$ .... Voir $k^{ème}$ activation
Période 18
Périodique 17
Préemptée 9
Prête 9
Réceptrice 11
Requête 18
Réveil 9
Réveillée 9
Simultanée 18
Sporadique 21
Supprimée 10
Suspendue 10
Tâche au repos 121
Tâche oisive 103
Taux d'utilisation 18
Taux de réaction 20
t_c.......... *Voir* Date du dernier temps creux acyclique
Temps creux 27
Temps creux acyclique 86
Temps creux cyclique 85
Temps d'inactivité .... *Voir* Temps creux
Temps de blocage maximal 53
Temps de réponse 19
Temps de retard admissible .... *Voir* Latence
Temps réel 5
A contraintes mixtes 17
A contraintes relatives 17
A contraintes strictes.. 17
TRCM .... *Voir* A contraintes mixtes
TRCR .... *Voir* A contraintes relatives
TRCS.......................... Voir A contraintes strictes
$terminaison_{i,k}$ 19
Théorème du reste chinois généralisé 156
Transition 65
Transition franchissable 66
Transition sensibilisée *Voir* Transition franchissable
TRCG ... *Voir* Théorème du reste chinois généralisé
$TR_{i,k}$ .... Voir Temps de réponse
$TxR_{i,k}$ .... *Voir* Taux de réaction
U_S .... *Voir* Charge d'un système
Vitesse maximale 76
W .... *Voir* Pondération du flot

INDEX DES FIGURES

figure I-1-1 : interactions entre procédé et contrôleur 6
figure I-1-2 : chronogramme du comportement d'un système temps réel synchrone 7
figure I-1-3 : chronogramme du comportement d'un système temps réel asynchrone : les événements entrants sont toujours visibles 8
figure I-1-4 : graphe des états possibles des tâches gérées par un noyau temps réel 10
figure I-1-5 : événement non mémorisé : cas où l'événement a lieu après la mise en attente 14
figure I-1-6 : événement non mémorisé : cas où l'évènement a lieu avant la mise en attente 14
figure I-1-7 : modèle initial d'une tâche temps réel 17
figure I-1-8 : chronogramme des activations et échéances d'une tâche périodique 19
figure I-1-9 : notations caractérisant l'exécution d'une tâche 20
figure I-1-10 : exemple de gigue temporelle d'une tâche 20
figure I-1-11 : chronogramme d'une tâche sporadique 22
figure I-1-12 : traitement d'une requête de tâche sporadique dans le serveur périodique associé 22
figure I-1-13 : valeurs limites des paramètres temporels d'un serveur périodique de tâche sporadique 23
figure I-1-14 : relation d'ordre définie sur la configuration matérielle du contexte 24
figure I-2-1 : séquence d'ordonnancement produite par Rate Monotonic 34
figure I-2-2 : séquence d'ordonnancement produite par Deadline Monotonic 36
figure I-2-3 : séquence d'ordonnancement produite par Deadline Monotonic 39
figure I-2-4 : séquence d'ordonnancement produite par Earliest Deadline 40
figure I-2-5 : Earliest Deadline / Least Laxity 41
figure I-2-6 : découpage d'une tâche autour des points de communication 43
figure I-2-7 : un graphe de précédence d'un système de tâches 43
figure I-2-8 : découpage d'un système de tâches en forme normale 44
figure I-2-9 : la tâche τ_1 est victime d'une inversion de priorité de la part de τ_2 à cause de τ_3 47
figure I-2-10 : séquence d'ordonnancement valide d'un système partageant une ressource 48
figure I-2-11 : exécution possible du même système 48
figure I-2-12 : résolution de l'inversion de priorité par le protocole à priorité héritée 49
figure I-2-13 : utilisation du protocole à priorités héritées 50
figure I-2-14 : utilisation du protocole à priorité plafond 51
figure I-2-15 : un interblocage 52
figure I-2-16 : interblocage évité par le PPP 52
figure I-2-17 : séquence valide pour le système S 55
figure I-2-18 : séquence au plus tôt pour le système S 56
figure I-2-19 : ordonnancement d'un système sur 2 processeurs : (a) est un ordonnancement valide, (b) est produit par ED 58
figure I-2-20 : ordonnancement d'un système sur 2 processeurs : (a) est un ordonnancement valide, (b) est produit par LL 58
figure I-2-21 : arbre de recherche d'ordonnancement dans le contexte $|1|r_i,C_i,D_i,noprmpt|$ 60
figure I-2-22 : technique d'énumération de placements à l'aide d'un arbre bi-colore 61
figure II-1-1 : réseau places-transitions 66
figure II-1-2 :réseau de Petri 66
figure II-1-3 : un réseau de Petri et un automate fini dont le langage terminal est $(ab)^*$ 68
figure II-1-4 : un réseau de Petri dont le langage terminal est a^nb^n 69
figure II-1-5 : un réseau de Petri coloré 70
figure II-2-1 : un réseau de Petri temporel 72
figure II-2-2 : graphe de classe d'un réseau de Petri temporel 73
figure II-2-3 : une machine à 2 registres modélisée par RdP temporel 75
figure II-2-4 : états successifs d'un RdP T-temporisés 76
figure II-2-5 : modélisation d'une machine à deux registres à l'aide d'un RdP T-temporisé 77
figure II-2-6 : un réseau de Petri avec conflits 78
figure II-2-7 : transformation d'un RdP T-temporisé à vitesse maximale en RdP autonome avec la règle de tir maximal, la place en pointillés est utilisée si l'on souhaite éviter la réentrance 79
figure II-2-8 : un réseau de Petri simple avec la règle de tir maximal 79
figure III-1-1 : séquence d'ordonnancement de S par ED, un temps creux existe 86
figure III-1-2 : charge induite par les requêtes au cours du temps 86
figure III-1-3 : temps creux acycliques pendant un ordonnancement multiprocesseur 88
figure III-1-4 : diagramme de charge correspondant 89
figure III-1-5 : modèle de communications considéré 92
figure III-1-6 : notations utilisées pour dénoter la charge précédant les attentes et envois de messages 93
figure III-1-7 : une attente directe 93

figure III-1-8 : illustration d'une attente indirecte 95
figure III-1-9 : diagramme de charge du système donné dans l'exemple III-1-3, le trait gras représente C_{att}..... 99
figure III-1-10 : séquence produite par earliest deadline pour l'exemple III-1-3 100
figure III-1-11 : diagramme de charge du système donné dans l'exemple III-1-4, le trait gras représente C_{att}..... 101
figure III-2-1 : diagramme de charge d'un système 104
figure III-2-2 : diagramme de charge de l'exemple III-2-2, la charge en attente est représentée en traits gras 106
figure III-2-3 : un ordonnancement valide d'un système de tâches $S=\{\tau_1<1,1,1,4>,\tau_2<0,2,4,4>,\tau_3<4,1,4,4>\}$ avec τ_1 et τ_2 partageant une ressource critique R. Il est nécessaire de placer le temps creux acyclique de façon non conservative pour que ce système soit ordonnançable. 107
figure III-2-4 : diagramme de charge du système $S=\{\tau_1<1,1,1,4>,\tau_2<0,2,4,4>,\tau_3<4,1,4,4>\}$. 107
figure III-2-5 : diagramme de charge correspondant à la séquence produite sur la figure III-2-3. 108
figure IV-1-1 : schéma d'ensemble de la modélisation par réseau de Petri 114
figure IV-1-2 : modèle pour une action périodique de période 8 115
figure IV-1-3 : partie horlogerie du modèle 115
figure IV-1-4 : la partie structurelle des tâches est la partie inférieure 116
figure IV-1-5 : modélisation du $j^{ème}$ bloc indépendant de τ_i, une double flèche sur un arc représente l'union de deux arcs : un dans chaque sens 117
figure IV-1-6 : modélisation d'un système de tâches par réseau de Petri 118
figure IV-1-7 : modélisation d'une communication par boîte aux lettres 119
figure IV-1-8 : modélisation de l'utilisation d'une ressource critique 119
figure IV-1-9 : accès à une ressource par un lecteur (à gauche) et un écrivain (à droite) 120
figure IV-1-10 : états du RdP modélisant un système d'une tâche indépendante $S=\{<0,3,3,3>\}$ 121
figure IV-1-11 : prise en compte des temps creux acycliques 122
figure IV-1-12 : modélisation de l'horlogerie d'une tâche retardée au-delà de sa période 123
figure IV-1-13 : schématisation du comportement temporel d'une tâche 124
figure IV-1-14 : modélisation d'un système de tâches interagissantes à l'aide d'un RdP, on ne représente pas $W(Processeur,T_{i,j,k})=W(T_{i,j,k},Processeur)=1$ 128
figure IV-1-15 : (a) un modèle réseau de Petri sans tâche oisive, (b) le même système avec tâche oisive 129
figure IV-1-16 : modélisation d'un système de trois tâches : τ_1 est non préemptible, les deux autres différées 131
figure IV-2-1 : deux visualisations d'un graphe d'accessibilité contenant l'ensemble des séquences d'ordonnancement valides d'un système de tâches 134
figure IV-2-2 : un graphe d'accessibilité contenant l'ensemble des séquences d'ordonnancement valides d'un système de tâches différées 135
figure IV-2-3 : un hyperpavé de dimension 3 et de cotes {3,4,2} 137
figure IV-2-4 : le GM de S_1 n'occupe que la moitié de l'hyperpavé le bornant 138
figure IV-2-5 : le GM de S_2 n'occupe qu'un tiers de l'hyperpavé le bornant 138
figure IV-2-6 : un arbre binaire pouvant représenter les entiers de 0 à 7 139
figure IV-2-7 : construction du graphe en largeur pour un système de 2 tâches 142
figure IV-2-8 : construction du graphe en profondeur pour un système de 2 tâches 143
figure IV-2-9 : gain apporté par l'utilisation de la table de latences lors de la construction d'un graphe des marquages pour le système de tâches indépendantes $S=\{\tau_1<0,3,5,5>, \tau_2<0,1,5,5>, \tau_3<0,2,10,10>\}$: le graphe des marquages est représenté par les flèches, les pointillés représentent ce qui est construit en sus en l'absence d'optimisation par la table de latence 146
figure IV-2-10 : gain apporté par l'utilisation des contraintes de successeur sur un système de 2 tâches indépendantes simultanées 147
figure IV-2-11 : recherche des séquences de $S=\{\tau_1<0,3,5,5>, \tau_2<0,1,5,5>, \tau_3<0,2,10,10>\}$ exécutant au plus tôt les tâches de $E_{opt}=\{\tau_2, \tau_3\}$ 151
figure IV-2-12 : minimisation du temps de réponse moyen des séquences d'ordonnancement d'un système $S=\{\tau_1<0,3,5,5>, \tau_2<0,1,5,5>, \tau_3<0,2,10,10>\}$ pour $E_{opt}=\{\tau_2, \tau_3\}$ 152
figure IV-2-13 : minimisation du temps de réponse maximal pour $E_{opt}=\{\tau_2, \tau_3\}$ 152
figure IV-2-14 : minimisation du taux de réaction moyen dans $S=\{\tau_1<0,3,5,5>, \tau_2<0,1,5,5>, \tau_3<0,2,10,10>\}$ pour $E_{opt}=\{\tau_2, \tau_3\}$ 153
figure IV-2-15 : minimisation du taux de réaction maximal dans $S=\{\tau_1<0,3,5,5>, \tau_2<0,1,5,5>, \tau_3<0,2,10,10>\}$ pour $E_{opt}=\{\tau_2, \tau_3\}$ 154
figure IV-2-16 : minimisation du retard moyen dans $S=\{\tau_1<0,3,5,5>, \tau_2<0,1,5,5>, \tau_3<0,2,10,10>\}$ pour $E_{opt}=\{\tau_2, \tau_3\}$ 154
figure IV-2-17 : Deux tâche urgentes mises en concurrence 156
figure IV-2-18 : valeurs interdites de r_j-r_i dans $\mathbb{Z}/p\mathbb{Z}$ avec $p=pgcd(P_i,P_j)$ 157
figure IV-2-19 : les valeurs possibles pour le choix de (r_j-r_i) pour éviter leur concurrence, avec $p=pgcd(P_i,P_j)$ 158

figure IV-2-20 : ordonnancements de deux tâches, deux séquences sont de gigue nulle 159
figure IV-2-21 : minimisation de la gigue de τ_1 dans $S=\{\tau_1<0,3,5,5>, \tau_2<0,1,5,5>, \tau_3<0,2,10,10>\}$avec un temps de réponse proche de 3 160
figure IV-2-22 : optimisation dans $S=\{\tau_1<0,3,5,5>, \tau_2<0,1,5,5>, \tau_3<0,2,10,10>\}$de la répartition des temps creux pour un premier temps de réponse de 2 161
figure V-2-1 : modélisation d'un système de tâches interagissantes à l'aide d'un RdP. On ne représente pas graphiquement le lien qui existe entre le processeur et chaque transition du système de tâches (i.e. $W(Processeur,T_{i,j,k})=W(T_{i,j,k},Processeur)=1$) 169
figure V-3-1 : un graphe d'accessibilité contenant l'ensemble des séquences d'ordonnancement valides d'un système de tâches de charge maximale sur 2 processeurs 172
figure V-3-2 : un graphe d'accessibilité contenant l'ensemble des séquences d'ordonnancement valides d'un système de tâches sur 2 processeurs 173
figure V-3-3 : séquences d'ordonnancement montrant que les préemptions entre tâches indépendantes sont nécessaires à l'ordonnançabilité d'un système 176
figure VI-1-1 : copie d'écran de QTSD...180
figure VI-1-2 : copie d'écran du joueur de jetons CPNS 181
figure VI-1-3 : copie d'écran de PeNSMARTS 182
figure VI-1-4 : options disponibles pour la construction du graphe des marquages...183
figure VI-1-5 : PeNSMARTS après construction du graphe des marquages 183
figure VI-1-6 : visualisation d'un diagramme de Gantt à l'aide de PLOT 184
figure VI-2-1 : fonctionnement de la mine..186
figure VI-2-2 : diagramme de contexte du contrôle de la mine...186
figure VI-2-3 : diagramme préliminaire du contrôle de la mine 187
figure VI-2-4 : préfixe d'un ordonnancement au plus tôt..190
figure I-2-1 : hiérarchie simplifiée des classes de complexité algorithmique 213
figure I-3-1 : un réseau de Petri avec la règle de tir maximal...215
figure I-3-2 : fonctionnement de l'algorithme de construction des ensembles maximaux de transitions franchissables..217
figure I-4-1 : activations de τ_i au cours du temps 219

TITLE: Optimal off-line scheduling of real-time systems by use of Petri nets on an uniprocessor and a multiprocessor architecture

ABSTRACT: We present an off-line methodology of validation of real-time systems by means of an enumeration technique of feasible schedules. The considered tasks are periodic, can be asynchronous, can communicate, share resources, and be compounded with non preemptible parts. We use a Petri net in order to model a real-time task system. Its language is the entire set of feasible schedules of the modeled system. The time is taken into account in the net by means of the earliest firing rule. A fundamental problem for the obtainment of this language was to get the depth of the state graph which is to be constructed. In the case of synchronous task system, this bound is trivial. But in the case of asynchronous task systems, we have shown that the depth of the graph is bounded too. This fundamental result in real-time scheduling implies that, whatever the scheduling policy is, the simulation duration of an asynchronous task system is bounded. Once the state graph is constructed, we use a linear time algorithm in the size of the state graph, based on a shortest paths search, in order to obtain optimal feasible schedules regarding some criteria. Each criteria reduces the size of the state graph, and the search for optimal schedules can be applied recursively in order to obtain interesting feasible schedules. Since the construction of the set of feasible schedules is exponential in space and time, we propose several optimizations and heuristics in order to reduce the time and space used. The uniprocessor and multiprocessor cases have been investigated.

KEYWORDS : cyclicity of schedules, simulation duration, asynchronous tasks, optimal scheduling, Petri nets under the earliest firing rule, jitter

RÉSUMÉ :

Les applications temps réel étant soumises à des contraintes temporelles, leur implantation nécessite, en plus d'une validation sémantique, une validation temporelle, c'est à dire une étude de l'ordonnancement des tâches sur le ou les processeurs. Tant que les tâches sont complètement préemptibles, il existe des algorithmes d'ordonnancement polynomiaux permettant d'ordonnancer des systèmes de tâches périodiques. Lorsque les tâches partagent des ressources critiques ou possèdent des parties non préemptibles, le problème de l'ordonnancement devient NP difficile. Un certain nombre d'optimisations et d'algorithmes approchés ont été mis en œuvre pour remédier à ce problème, mais ils n'ont pas toujours un comportement désirable. De plus, il n'existe pas de moyen analytique (i.e. hors de la simulation dite hors-ligne de l'ordonnancement des tâches) de valider temporellement de tels systèmes. La démarche adoptée dans cette thèse consiste à utiliser une approche exhaustive de l'ordonnancement. Les tâches d'un système sont modélisées par un réseau de Petri (RdP) dont le langage est l'ensemble de tous les ordonnancements possibles des tâches. La notion de temps y est introduite non pas par une temporisation (au sens temporel ou temporisé des RdP) mais par un fonctionnement avec la règle de tir maximal d'un RdP classique. Tous les ordonnancements sont donc obtenus par construction du graphe des marquages du RdP.

Un des premiers problèmes pour l'obtention de ce graphe est de déterminer sa profondeur, c'est à dire jusqu'à quelle profondeur aller pour que les séquences obtenues soient cycliques. Dans le cas où les tâches démarrent au même instant, la profondeur est triviale à calculer. Dans le cas contraire, nous avons montré que la profondeur était bornée pour toute séquence d'ordonnancement. Ce résultat fondamental pour la théorie de l'ordonnancement permet d'appliquer la plupart des techniques hors-lignes au cas où les tâches périodiques ne démarrent pas toutes au même instant, alors que ces techniques se limitaient jusqu'ici au cas de tâches périodiques simultanées.

Une fois le graphe des marquages construit, on peut extraire, en un temps linéaire de la taille du graphe, des séquences d'ordonnancement optimales suivant certains critères à l'aide d'algorithmes de recherche de plus courts chemins. Chaque critère à optimiser réduit la taille du graphe, sur lequel on peut appliquer d'autres critères d'optimisation, jusqu'à obtenir un ensemble de séquences que l'on juge satisfaisantes. Cette approche étant exponentielle en temps et en espace, une grande partie du travail de recherche consiste à stopper au plus tôt la construction de certaines branches du graphe des marquages menant à des séquences d'ordonnancement peu intéressantes. Nous utilisons à cette fin des techniques permettant de limiter le nombre de changements de contexte des tâches à l'aide de contraintes de successeur sur les transitions : cela a pour effet de supprimer des maillages du graphe pour n'en conserver que les arêtes externes. Nous utilisons aussi des contraintes absolues sur les temps de réponse désirés pour les tâches.

L'étude a été menée dans les contextes monoprocesseur et multiprocesseur.

SECTEUR DE RECHERCHE : INFORMATIQUE

MOTS-CLÉS : cyclicité des ordonnancements, durée de simulation, tâches différées, ordonnancement optimal, réseaux de Petri avec la règle de tir maximal, gigue

INTITULÉ ET ADRESSE DE L'U.F.R OU DU LABORATOIRE :

LISI-ENSMA
Téléport 2
1, avenue Clément Ader
BP 40109
86961 Futuroscope Chasseneuil Cedex