UNIVERSITÉ DE POITIERS
Faculté des Sciences Fondamentales et Appliquées
ENSMA : *École Nationale Supérieure de Mécanique et d'Aérotechnique*
ÉCOLE DOCTORALE : *Sciences Pour l'Ingénieur & Aéronautique*

THÈSE

pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITÉ DE POITIERS

ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE

Discipline : **INFORMATIQUE**

présentée par **Stéphane PAILLER**

le 19 Octobre 2006

Analyse Hors Ligne d'Ordonnançabilité d' Applications Temps Réel comportant des Tâches Conditionnelles et Sporadiques

Directeur de thèse : **Annie Choquet-Geniet**

Équipe d'accueil : *Équipe Temps Réel du LISI*

Membres du Jury

Président	Michel Mériaux,	Professeur, Université de Poitiers
Rapporteur	Marc Bourcerie ,	Professeur, Université d'Anger
Rapporteur	Guy Vidal-Naquet,	Professeur, Supélec
Examinateur	Annie Choquet-Geniet,	M. de Conf. HdR, Université de Poitiers
Examinateur	Michel Augeraux ,	Professeur, Université de La Rochelle

Laboratoire d'Informatique Scientifique et Industrielle
École 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

Cette thèse est le fruit d'un travail mené au sein du Laboratoire d'Informatique Scientifique et Industrielle dirigée 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 souhaite également remercier le Professeur Francis Cottet, qui m'a permis d'intégrer l'équipe Temps Réel.

Je tiens à exprimer mes plus sincères remerciements à Annie Choquet-Geniet pour avoir encadré mon travail de thèse. Ses conseils et sa persévérance m'ont été une aide inestimable et ont largement contribué à ma formation.

Je souhaite remercier Guy Vidal-Naquet et Marc Bourcerie qui ont eu la lourde tâche de rapporter ma thèse, ainsi que les autres membres du jury, Michel Mériaux et Michel Augeraux qui m'ont fait l'honneur d'accepter d'être examinateurs.

J'adresse un remerciement tout particulier à Emmanuel Grolleau et Pascal Richard pour les discussions fructueuses sur le plan scientifiques, que nous avons eues.

Enfin, je remercie sincèrement tous les membres (et ex-membres) du laboratoire, JCP, Tex, Dago, Ricou, Micky, Mickey, Ridou, Yamine, Ladjel, Jéjé, JM, Laurent G, Dave, Laurent D., Faby, Karim, Manu G. pour les bons moments que nous avons passés ensemble et la bonne ambiance qu'ils font régner au LISI.

Mes derniers remerciements et non les moindres iront à mes proches, et en particuliers à Lucie et l'*Orphéas Team*, qui m'ont toujours apporté leur soutien sans faille.

À ma Lucie,

Table des matières

Introduction générale

Qu'ils s'agissent de véhicules, de robots ménagers, d'unités de production, de multimédias... nous sommes de plus en plus souvent confrontés à la gestion informatisée des appareils au sens large dont l'utilisation ponctue notre quotidien. Ceci a comme conséquence immédiate un développement intense des techniques de développement des systèmes réactifs (ou système de contrôle de procédés). Ces systèmes prennent de plus en plus le pas sur les technologies purement mécaniques, et jouent donc un rôle dont l'importance est en perpétuelle expansion. La première raison d'être de cette mutation relève d'un impératif de renforcement de la sécurité : les procédés contrôlés (qu'il s'agisse d'un système embarqué, par exemple dans un véhicule, ou d'un système de gestion centralisée, par exemple d'une centrale nucléaire) sont sensibles, tout dysfonctionnement peut s'avérer catastrophique. La sécurité doit pouvoir, à terme, être pleinement assurée par le système de contrôle, dont nous sommes en droit d'attendre des réactions plus rapides que ne le sont les réactions des organes mécaniques ou humains, et qui par ailleurs ne sont pas sujettes aux erreurs d'appréciation... La sécurité relève donc dans une large mesure de la fiabilité du système que gère le procédé : fiabilité fonctionnelle (les réactions aux évènements sont conformes au cahier des charges) tout autant que fiabilité opérationnelle (les évènements sont perçus et pris en compte à temps).

Un autre objectif poursuivi par les constructeurs est d'offrir aux utilisateurs un plus grand confort d'utilisation, ainsi que des services nouveaux (gestion totalement automatisée de la cuisson d'un four, système de navigation dans un véhicule...). Tout ceci nécessite la mise en place de plate formes de développement performantes, qui traitent d'une part des aspects fonctionnels, et d'autre part des aspects opérationnels. C'est ce second point qui nous intéresse dans le présent mémoire. La validation opérationnelle d'une application sous-entend la nécessité de procéder à des vérifications temporelles quantitatives. La sécurité nécessite que l'on puisse garantir qu'entre toute évolution du procédé et la commande produite en réaction par le système, une durée adaptée s'écoulera. En d'autre termes, il s'agit de garantir que l'informatique va au moins aussi vite que le procédé ! Cette propriété doit être prouvée avant la mise en exploitation de l'application. Cette application consiste en un ensemble de tâches, chacune d'elles ayant vocation à traiter une information reçue du procédé (la réception de l'information pouvant être assimilée à l'évènement) ou à réagir

à un évènement spécifique. L'application est donc décrite comme un ensemble d'activités, décrites par des tâches, qui doivent s'exécuter de manière répétitive tout au long de la vie de l'application.

Par ailleurs, l'application s'exécute sur une architecture souvent dédiée, qui peut être monoprocesseur, multiprocesseur ou répartie. Dans ce mémoire, nous considérerons des systèmes monoprocesseur. L'activité des processeurs doit être répartie entre les différentes tâches. De manière très classique, la répartition est à la charge d'un ordonnanceur. La spécificité vient ici du fait que l'objectif premier (voire même unique) de l'ordonnanceur est de garantir le respect des contraintes temporelles induites par le procédé. Par garantir, il faut prouver ; pour prouver, il faut formaliser. Il est donc nécessaire de modéliser l'application, et dans le cas qui nous préoccupe, c'est une modélisation temporelle de l'application qui s'impose. Une telle modélisation est utilisée depuis très longtemps dans la littérature, et la majeure partie (pour ne pas dire l'intégralité) des travaux sur l'ordonnancement s'appuient sur une modélisation souvent appelée modélisation de Liu Layland. Il s'agit de proposer une abstraction temporelle de fonctionnement de l'application. Une application consiste en un ensemble de tâches, chacune de ces tâches est décrite à l'aide de paramètres temporels qui peuvent indiquer (le cas échéant) l'instant de création de la tâche, sa période (s'il s'agit d'une tâche de contrôle, par essence même périodique), son délai critique (qui quantifie la "durée adaptée" entre la création d'une instance et sa terminaison), sa durée. Parmi tous ces paramètres, les trois premiers sont issus du cahier des charges, et dépendent du concepteur de l'application. Mais le dernier dépend directement du code de la tâche. Comme il est difficile (voire même dénué de sens) de déterminer une durée d'exécution unique et exacte, nous travaillons en fait avec un majorant de la durée (WCET). Mais cette approche "pire cas" masque la réalité des choses. Quand les variations de durées sont dues à l'indéterminisme du processeur, de la gestion mémoire, etc... il est difficile de faire autrement. Mais cette vue de durée unique masque également les variations liées au code lui même : quand la tâche comporte des instructions conditionnelles, nous ne pouvons plus parler de durée, mais plutôt de duréeS. Les études réalisées en masquant les instructions conditionnelles sont alors d'une part sur-contraintes (particulièrement dans le cas où l'application manipule des ressources) et d'autre part incapables de décrire le fonctionnement réel de l'application. Nous ne disposons plus d'outils de simulation. L'objectif de nos travaux est de tenter de reprendre le problème de l'ordonnancement dans le cas où certaines tâches comportent des instructions conditionnelles.

Nous définissons pour cela un nouveau modèle temporel de tâche qui étend celui issu de Liu Layland. Ce nouveau modèle tient explicitement compte des instructions conditionnelles présentes dans le code des tâches et permet ainsi de prendre en compte l'ensemble des durées d'exécution des tâches et le comportement réel de l'application vis à vis de la gestion des ressources. Nous sommes ainsi amenés à redéfinir le problème de l'ordonnancement dans le contexte des tâches conditionnelles. Nous mettons en évidence l'ordonnançabilité locale qui permet d'avoir une vision indépendante de chaque durée des tâches conditionnelles et l'ordonnancement global qui tient compte plus précisément de la parité des instructions conditionnelles et des comportements qu'elle engendre. Nous montrons que ces deux notions ne sont pas équivalentes dans un contexte d'applications partageant des ressources. Nous nous sommes également intéressés à la durée minimale de simulation nécessaire à la validation d'applications Temps Réel comportant des tâches

conditionnelles. Nous proposons un outil basé sur une modélisation par Réseau de Petri permettant d'intégrer notre nouveau modèle temporel. Cette modélisation a permis également d'étendre notre étude aux tâches sporadiques, dont l'activation est déclenchée par l'un des comportements d'une tâche conditionnelle.

Dans un premier chapitre nous exposons la problématique Temps Réel au travers des approches synchrones et asynchrones. Nous définissons plus précisément l'architecture des systèmes Temps Réel ainsi que les différentes notions de tâches et de leur qualité de service.

Dans un deuxième chapitre, nous exposons la problématique de l'ordonnancement et de la validation. Nous mettons en évidence les deux grands courants en ligne et hors ligne.

Le troisième chapitre expose une méthode qui repose sur une modélisation par RdP pour énumérer les séquences d'ordonnancement valides d'une application Temps Réel.

Le chapitre quatre présente le nouveau modèle Temps Réel de tâches tenant compte des instructions conditionnelles. Nous redéfinissons la problématique de l'ordonnancement et mettons en évidence l'ordonnançabilité locale et globale.

Enfin, la cinquième partie, reprend la modélisation par RdP pour l'énumération des séquences d'ordonnancement et l'étend à la prise en compte des tâches conditionnelles. Nous ajoutons également la modélisation de tâches sporadiques dont l'exécution est déclenchée par l'un des comportements d'exécution des tâches conditionnelles. Puis nous expliquons comment nous obtenons des graphes d'ordonnancement à partir de cette modélisation.

Première partie

État de l'art

Chapitre Premier

Les Systèmes Temps Réel

Chapitre Premier

Les Systèmes Temps Réel

Cette première partie introduit la problématique temps réel. Nous décrivons plus particulièrement l'approche asynchrone des applications temps réel. Après avoir vu les caractéristiques de l'architecture des systèmes temps réel, nous présentons les éléments permettant d'appréhender une étude temporelle des applications temps réel. Nous pouvons ainsi introduire les différents modèles de tâches temps réel composant ce type application.

Sommaire

Chapitre Premier

Les Systèmes Temps Réel

Chaque Système Informatique(isé) possède ses propres spécificités, étroitement dépendantes de son domaine d'utilisation. Toutefois, nous pouvons regrouper les applications informatiques parmi les trois catégories suivantes [Ell97]. Nous qualifions de :

Transformationnelles, toutes les applications permettant d'élaborer un résultat à partir de données connues et disponibles à l'initialisation de l'application. Leur traitement effectif est de plus non contraint dans la durée. Les applications de calcul scientifique ou de gestion de bases de données sont des exemples représentatifs de cette catégorie d'application.

Interactives, toutes les applications permettant de fournir des résultats en fonction de données produites par l'environnement du système (essentiellement imputables à l'utilisateur), dans un délai de production satisfaisant des valeurs moyennes statistiques. Parmi ces applications, les progiciels de bureautique sont certainement les plus connus.

Réactives, toutes les applications dont les résultats sont entièrement liés à l'environnement qui leur est connecté. De plus, la dynamique de cet environnement conditionne les instants de production de ces résultats.

Cette dernière catégorie est souvent assimilée aux Systèmes Temps Réel puisque d'une part, elle suggère par le terme *réactif*, les deux paradigmes essentiels aux Systèmes Temps Réel : l'*interactivé* et le *temps* et d'autre part l'émergence de l'informatisation du contrôle de procédés de plus en plus complexes contribue à une utilisation en forte croissance de ce type de systèmes. Les domaines concernés vont des chaînes de production [RCK01, KS99, Kai01] aux transports aériens en passant par des robots de plus en plus perfectionnés, tels que les modules d'exploration planétaire (Pathfinder [CDKM00]), l'aide à la conduite automobile ou aux applications multimédia contraintes par le temps [IFS04]. Si le caractère réactif prédomine dans ces exemples, la criticité du *temps* n'en est pas moins important pour ces systèmes. En effet, alors que le retard de quelques millisecondes ou la perte d'une trame vidéo sur un réseau ne nuira qu'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 [Sta88].

1.1 Définitions et Taxinomie

1.1.1 Définitions

Il existe de nombreuses définitions des *Systèmes Temps Réel*. Une première définition tirée de [Sta88], qualifie de système Temps Réel :

«tout système informatique dont le bon fonctionnement ne dépend pas uniquement de la correction algorithmique et logique mais également des dates d'arrivée des résultats».

Parmi les systèmes réactifs, la différentiation des systèmes Temps Réel est ici mise en évidence par la notion d'*erreur* (failure) qui prend également son sens avec le retard de la production de résultats. Toutefois, si la notion de correction temporelle est bien mise en évidence, le caractère réactif n'est pas explicitement défini. Une deuxième approche tente de définir les systèmes Temps Réel comme étant :

«des systèmes ouverts répondant constamment aux sollicitations de leur environnement en produisant des actions sur celui-ci».

Ici, nous percevons la notion de servitude vis à vis du procédé contrôlé mais l'aspect temporel n'est qu'évoqué. C'est au CNRS [CNR88] que nous pouvons enfin trouver la définition suivante qui allie les deux paradigmes primordiaux des Systèmes Temps Réel :

«Peut être qualifiée de temps réel toute application mettant en œuvre un système informatique dont le fonctionnement est assujetti à l'évolution dynamique de l'état d'un environnement(procédé) qui lui est connecté et dont il doit contrôler le comportement».

Pour affiner cette définition en introduisant les notions de critères temporels, nous pouvons citer la définition donnée par [AD92] :

«Une application temps réel constitue un système de traitement de l'information ayant pour but de commander un environnement imposé en respectant les contraintes de temps et de débit (temps de réponse à un stimulus, taux de perte d'information toléré par entrée) qui sont imposées à ses interfaces avec cet environnement».

1.1.2 Applications Temps réel

Les définitions précédentes des Systèmes Temps Réel ont mis en évidence deux éléments distincts : une ou plusieurs entités physiques constituant le procédé, dont le rôle est d'agir et de détecter, et un contrôle informatique, nommé *contrôleur* ou *application temps réel* qui est le décideur des actions (ou réactions) 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 1.1 donne un aperçu des interactions qui existent entre procédé et contrôleur d'un système temps réel.

1.1.2.1 Applications concurrentes

La dynamique des périphériques (ou interfaces) du procédé et de son environnement dicte celle de l'interaction entre le procédé et le contrôleur. De la même façon que l'environnement évolue en parallèle avec les périphériques du système, le contrôleur, c'est à dire l'application temps réel, doit pouvoir refléter ce parallélisme puisqu'il doit connaître

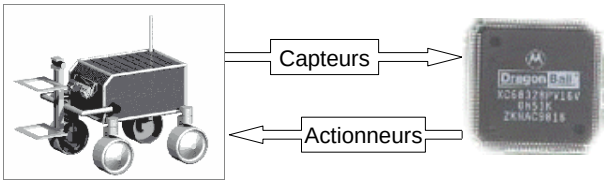

Fig. 1.1: Un système Temps Réel : interaction procédé-contrôleur.

l'état de l'environnement par l'intermédiaire des capteurs et piloter le procédé par les actionneurs.

Pour gérer toutes ces entités interagissantes, il est donc nécessaire de développer des techniques logicielles permettant de traiter les informations issues des capteurs sur l'unité de calcul pour produire les actions adéquates.

Pour cela, nous utilisons le principe des *applications concurrentes* dont nous pouvons trouver une définition dans [BW90] :

«La programmation concurrente est le nom donné aux techniques et notations de programmation pour exprimer le potentiel parallèle et résoudre les problèmes inhérents de synchronisation et de communication. L'implémentation du parallélisme est un problème des systèmes informatiques (logiciel ou matériel) qui est avant tout indépendant de la programmation concurrente. La programmation concurrente est importante puisqu'elle permet de faire abstraction de l'étude du parallélisme sans être confronté aux détails de l'implémentation».

Cette approche est particulièrement prometteuse en ce qu'elle permet d'augmenter la puissance d'expressivité et de réduire le coût de développement par rapport à une application fonctionnant sans parallélisme. En effet, sans concurrence, l'implémentation de telles applications nécessiterait l'utilisation d'une boucle de contrôle permettant de gérer toutes les interactions possibles même si celles-ci ne sont pas utiles. De plus, cette boucle de contrôle ne permettrait pas de conserver la distinction logique entre les différents composants du procédé et augmenterait ainsi la difficulté de gestion des contraintes temporelles. Toutefois, l'utilisation des applications concurrentes n'est pas sans inconvénients. En effet, il devient nécessaire de fournir un *run-time* ou *noyau temps réel* pour superviser l'exécution sur l'unité de calcul des différentes tâches du système.

1.1.2.2 Notion de Tâche

La notion de capteurs et d'actionneurs introduit implicitement, du point de vue logiciel, l'utilisation de différentes *tâches* permettant de les piloter. Ce sont des programmes séquentiels dédiés au traitement d'un des composants du systèmes Temps Réel. Par exemple un programme Temps Réel peut être constitué d'une collection de tâches telles que :

- des exécutions périodiques de mesures de différentes grandeurs physiques (pression, température, accélération etc...). Ces valeurs peuvent être comparées à des valeurs de consignes liées au cahier des charges du procédé ;
- des traitements à intervalles réguliers ou programmés ;

– des traitements en réaction à des évènements internes ou externes ; dans ce cas les tâches doivent être capables d'accepter et de traiter en accord avec la dynamique du système les requêtes associées à ces évènements.

Nous caractérisons ainsi une *Application Temps Réel* comme étant une application *multi-tâches*.

1.1.2.3 Interactions entre les tâches

Les tâches, dont les comportements sont séquentiels, peuvent interagir entre elles pour assurer le bon fonctionnement global de l'application que le système pilote. Il est donc nécessaire de fournir parallèlement à ces tâches des moyens de communication et de synchronisation permettant de gérer tous les problèmes liés aux accès à des ressources communes comme par exemple les périphériques (terminaux, imprimantes *etc...*), ou l'exécution des tâches ordonnées par des critères de précédence.

Dans le cas du partage de ressources, certaines d'entre elles peuvent être bornées en nombre d'accès simultanés. Dans ce cas, nous parlons de *ressources critiques*. Pour permettre un bon fonctionnement de l'application, il est nécessaire de mettre l'accès à ces ressources en *exclusion mutuelle*. Il faut s'assurer qu'il y ait bien au plus le nombre maximum autorisé de tâches simultanément en *section critique*, c'est à dire qui utilisent simultanément la ressource. De plus, il convient de garantir la non préemption des ressources en cours d'utilisation. L'accès à ces ressources peut de plus s'effectuer en mode *lecture* ou *écriture*, chacun possédant son propre nombre d'accès simultanés autorisés.

Les critères de précédence des tâches sont pour la plupart issus soit d'un désir d'échange de données entre deux tâches, soit de la volonté de synchroniser deux tâches pour que la suite de leur exécution s'effectue en parallèle par un mécanisme de Rendez-Vous. Dans le premier cas, on identifie une tâche émettrice et une tâche réceptrice.

Nous parlons de *tâches indépendantes* lorsque l'application n'utilise ni ressources critiques, ni synchronisation.

1.2 Architecture des Systèmes Temps Réel

1.2.1 Architecture matérielle

Les systèmes temps réel peuvent être classés selon leur couplage avec des éléments matériels avec lesquels ils interagissent. Ainsi, l'application concurrente et le système d'exploitation qui lui est associé peuvent se trouver :

– soit directement dans le procédé contrôlé : c'est ce que l'on appelle des *systèmes embarqués* (embedded systems). Le procédé est souvent très spécialisé et fortement dépendant du calculateur. Les exemples de systèmes embarqués sont nombreux : contrôle d'injection automobile, stabilisation d'avion, électroménager.... C'est le domaine des systèmes spécifiques intégrant des logiciels sécurisés optimisés en encombrement et en temps de réponse.
– soit le calculateur est *détaché* du procédé : c'est souvent le cas lorsque le procédé ne peut être physiquement couplé avec le système ou dans le cas général des contrôle/commandes de processus industriels. Dans ce cas, les applications utilisent

généralement des calculateurs industriels munis de systèmes d'exploitation standards ou des automates programmables industriels comme dans les chaînes de montage industrielles par exemple.

En introduisant la notion de calculateur ou de processeur, nous distinguons trois grandes catégories d'architecture matérielle pour les Systèmes Temps Réel en fonction de leur richesse en terme de nombre de cartes d'entrée/sortie, de mémoires, de processeurs et de la présence de réseaux.

- L'architecture *monoprocesseur* : un unique processeur exécute toutes les tâches de l'application concurrente. Dans ce cas, la notion de parallélisme n'a plus vraiment de sens puisque le temps processeur est partagé entre toutes les tâches. Nous parlons plutôt de *pseudo-parallélisme* ou d'*entrelacement* des exécutions. En effet, le parallélisme des tâches semble réel à l'échelle de l'utilisateur mais le traitement sur l'unique processeur s'opère de façon séquentielle.
- L'architecture *multiprocesseurs* : l'exécution de toutes les tâches est ici répartie sur n processeurs partageant une unique mémoire centrale. La coopération entre tâches se fait par partage des informations placées en mémoire. Le traitement est donc ici réellement parallélisé.
- L'architecture *distribuée* : c'est le cas des architectures multiprocesseurs ne partageant pas de mémoire centrale. Ces processeurs sont reliés entre eux par l'intermédiaire de réseaux permettant d'assurer les communications entre les différentes tâches. Une ferme d'ordinateurs est un exemple typique de cette architecture. La coopération se fait ici par communication par réseau.

Par la suite nous nous placerons dans le cas des architectures monoprocesseur.

1.2.2 Environnement d'exécution

Nous venons d'explorer les différentes formes que peuvent prendre les Systèmes Temps Réel. La figure 1.2 montre plus particulièrement celle qui nous intéresse pour la suite de cette étude.

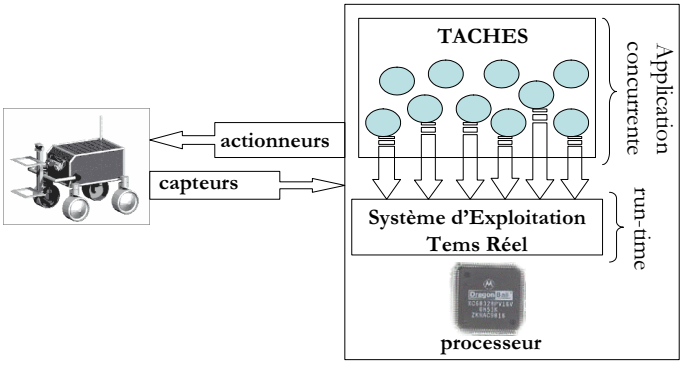

Fig. 1.2: Structure générale de l'architecture monoprocesseur d'un application Temps Réel.

Si cette architecture permet de mettre en avant les qualités réactives de ces systèmes,

il n'en va pas de même pour la seconde notion primordiale : le temps. Il est vital pour le procédé de pouvoir garantir que le traitement des différents évènements interviendra dans des délais adaptés à l'évolution du procédé. Pour cela, il est nécessaire d'associer à un système réactif des contrôles temporels. Ceux-ci doivent évaluer la vitesse de réaction du système de contrôle. Deux approches peuvent être envisagées : l'approche *synchrone* et l'approche *asynchrone*. Dans le premier cas, le respect des délais est implicite. Dans le deuxième cas, il devra être contrôlé. De plus, il sera nécessaire de se placer dans un environnement d'exécution adapté, qui sera fourni par un *noyau Temps Réel* ou *exécutif Temps Réel*, proposant des routines spécifiques temporellement documentées.

1.2.2.1 Synchrone versus Asynchrone

Une première approche consiste à considérer que le temps de traitement des différentes tâches qui composent l'application concurrente est de durée nulle [Ell97]. En effet, tout évènement ou interruption arrivant dans le système est associé à une tâche et le traitement de celle-ci nécessite un masquage des interruptions jusqu'à sa terminaison. Or, si ce temps de traitement est non nul, d'autres évènements peuvent arriver et n'être pris en compte qu'au bout d'une certaine durée (une fois le précédent traitement achevé). Ce retard correspond précisément au critère que nous souhaitons optimiser. De plus, dans le cas où plusieurs interruptions n'ont pas été traitées pendant l'exécution d'une tâche, il devient nécessaire d'établir des règles permettant de choisir la prochaine tâche à exécuter. L'utilisation d'une durée de traitement nulle permet ainsi de contourner toutes ces difficultés. C'est l'hypothèse d'instantanéité qui prend tout son sens dans certain système dédié au contrôle-commande où les processus ont une vitesse de traitement suffisamment grande comparée à la dynamique de l'environnement du procédé. Ainsi, il est logique de faire l'hypothèse qu'il est inutile d'interrompre le traitement d'une tâche par une autre, jugée plus prioritaire pour la sauvegarde du système, puisque la durée effective de traitement des tâches est négligeable donc considérée comme nulle. Ces systèmes sont donc *non préemptifs*. La programmation de cette approche est appelée la programmation *synchrone*. Elle utilise des *langages synchrones* comme *Esterel* [BG92] et *Lustre* [HCRP91] par exemple. Un atout immédiat de l'approche synchrone est sa simplicité de mise en œuvre puisqu'il existe de nombreux outils automatiques de génération du code ainsi que des outils formels de vérification des propriétés comme la vivacité ou la sûreté des programmes produits. Nous pouvons citer les techniques basées sur le *model checking* [Rou99, CES86] par exemple.

Toutefois, l'hypothèse d'instantanéité n'est pas applicable à toutes les applications puisqu'elle néglige à la fois du problème lié au masquage des interruptions durant l'exécution d'une tâche et celui du retard entre une interruption et la fin de traitement de la tâche qui lui est associée. En effet, les systèmes temps réel sont souvent complexes et peuvent comporter un grand nombres de tâches engendrant une forte occupation du processeur et la dynamique des procédés se rapprochent davantage de celle des systèmes de contrôle. De plus, les industriels sont souvent confrontés à des critères de coût les incitant à utiliser des processeurs moins performants augmentant de fait la durée de réaction à un évènement, c'est à dire la durée entre l'occurrence d'une interruption et la terminaison de la tâche qui lui est associée.

Dans les cas où l'approche synchrone ne peut pas être utilisée, une technique de mise en œuvre *asynchrone* est choisie. Celle ci consiste à mettre en place un mécanisme d'écoute des occurrences d'évènements même pendant l'exécution d'une tâche sur le processeur. De plus, la tâche en exécution peut être interrompue à tout moment au bénéfice d'une autre tâche jugée plus urgente. On appelle *préemption*, le mécanisme d'interruption de l'exécution d'une tâche au profit d'une autre. L'approche asynchrone considère donc que les tâches possèdent une durée d'exécution non nulle puisque l'exécution est "découpable". Deux constatations peuvent être faites :

- L'introduction d'une durée d'exécution entraîne l'asynchronisme entre l'occurrence d'un évènement et son traitement, même si la notion de préemption est utilisée. Ceci se traduit par l'asynchronisme entre le fonctionnement du procédé et les évènements induits par l'environnement.
- L'approche asynchrone oblige à quantifier le retard entre l'occurrence d'un évènement et la terminaison de l'exécution de la tâche qui lui est associée. Cette valeur permet d'évaluer la réactivité du Système Temps Réel en fonction de la dynamique de l'environnement du procédé. On introduit ainsi un paramètre permettant de mesurer la réactivité de l'exécution des tâches, par une borne de temps ou *échéance* attribuée à chaque tâche. Ce paramètre permet de s'assurer du comportement Temps Réel de l'application concurrente et donc du procédé vis à vis des sollicitations extérieures.

Cette approche conduit à la nécessité de disposer d'un système d'exploitation temps réel permettant de gérer l'ensemble des tâches de l'application concurrente. Cette gestion consiste d'une part, à mesurer le temps absolu pour pouvoir ainsi se rendre compte du respect des échéances des tâches et d'autre part, décider de l'ordre d'exécution des tâches pour palier le retard dû à l'asynchronisme. Nous nous intéressons par la suite uniquement à cette approche asynchrone.

1.2.2.2 Exécutif Temps Réel

Nous pouvons maintenant décomposer l'architecture logicielle d'un système Temps réel en deux couches. La première consiste en une application concurrente composée d'un ensemble de tâches. Nous utilisons également le terme d'*application multitâches*. La deuxième, de plus bas niveau, joue le rôle d'un système d'exploitation minimal chargé de faire le lien entre le procédé physique et l'application multitâches. Ce système d'exploitation, appelé *exécutif Temps Réel*, est de par la considération de l'asynchronisme, dirigé par les évènements, ceux-ci pouvant provenir de différentes sources :

- du procédé physique par l'intermédiaire d'interruptions matérielles associées à chaque évènement.
- du temps : chaque système est muni d'une horloge Temps Réel pouvant générer des interruptions.
- de l'application multitâches lorsque par exemple l'exécution d'une tâche est conditionnée par l'exécution d'autres tâches. Dans ce cas il faut que l'exécutif retarde l'exécution de cette tâche pour permettre au préalable au processeur d'exécuter les autres.

L'exécutif temps réel propose différents services et garanties facilitant l'exécution et la communication des tâches. Ces services appelés *primitives temps réel* peuvent être directement utilisés dans les tâches et sont de différentes natures :

gestion des tâches.

Celles ci changent d'état au cours de leur utilisation dans le système. Elles sont toutes initialement inexistantes dans le système. Elles sont alors "créées" puis réveillées ce qui les positionnent dans l'état "prête". Un mécanisme logiciel de choix décide alors d'élire une tâche parmi celles dans l'état "prête" pour que le processeur la traite. Dans ce cas l'état de la tâche passe à "exécutée". De cet état, une tâche peut soit être préemptée par une autre tâche, dans ce cas elle retourne dans l'état "prêt", soit être bloquée par une synchronisation, ce qui la fait passer à l'état "attente", soit enfin elle termine son exécution et passe dans l'état "terminée" avant de disparaître du système. La figure 1.3 résume ces différentes étapes.

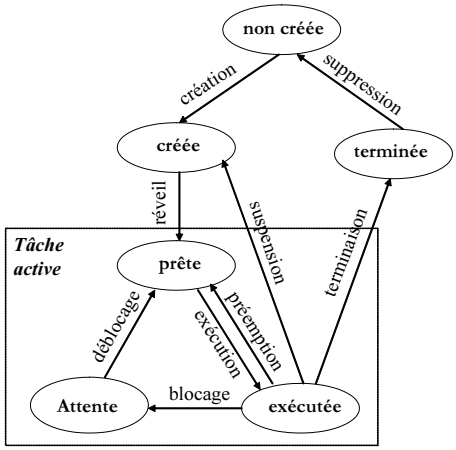

Fig. 1.3: Évolution possible des états des tâches dans un Système Temps Réel.

gestion des ressources partagées.

Nous avons vu que certaines ressources peuvent être critiques et qu'elles doivent alors être utilisées en exclusion mutuelle. L'utilisation de ces ressources nécessite des techniques permettant de garantir le respect de l'exclusion mutuelle. Par exemple, la plus simple consiste à masquer les interruptions durant l'utilisation des ressources, ce qui empêche l'exécutif temps réel de traiter les nouvelles demandes d'accès à une ressource et résoud du même coup les problèmes d'exclusion mutuelle. Toutefois, cette technique montre vite ses limites puisque l'utilisation d'une ressource peut être relativement longue et il n'est pas toujours souhaitable d'interdire la préemption (conséquence du masquage des interruptions) sur une telle durée. C'est pourquoi on lui préfère le plus souvent l'utilisation de *sémaphores*, qui permettent d'implémenter toutes sortes de politique d'accès à une ressource comme par exemple la politique FIFO (First In First Out) ou encore la politique de priorités fixes. Une fois l'exclusion assurée, il reste à la charge de l'exécutif temps réel de vérifier qu'il n'y a

pas de phénomène d'*interblocage*[1]. Pour cela, il existe des algorithmes et solutions relativement complexe reposant sur deux approches [Kra85] :

- la prévention : il s'agit d'algorithmes garantissant l'absence d'interblocages. L'algorithme du banquier en est un exemple. Il s'applique dans le cas où les comportements des tâches sont connus (on sait donc de quelles ressources elles auront besoin).
- la détection et la guérison : dans ce cas, l'exécutif temps réel doit être capable de détecter un interblocage lorsqu'il a lieu ce qui peut également être fait par l'algorithme du Banquier, et un algorithme dit de guérison doit alors permettre de rétablir le système dans un état stable où toutes les tâches sont débloquées. Pour cela, il est possible par exemple de reprendre des ressources à certaines tâches, quitte à les obliger à reprendre tout ou partie de leur exécution ce qui n'est toutefois pas envisageable dans le contexte d'applications soumises à des contrôles temporels. De plus, certaines techniques de guérison entraîne une perte d'information.

Bien évidement, dans le cas des exécutifs temps réel, la prévention est beaucoup plus appropriée puisque nous garantissons que le phénomène ne se produira jamais ce qui rend le système plus sûr.

gestion des communications entre tâches.

Il existe principalement deux types de communications entre tâches. La première dite asynchrone utilise le concept de *boîte aux lettres*. Ce dernier est fondé sur l'utilisation d'un tampon d'échange de données : une tâche émettrice dépose chaque donnée dans ce tampon qui les gère en mode FIFO ; la tâche réceptrice, lorsqu'elle a besoin de la donné, doit soit se mettre en attente si le tampon est vide, soit recueillir la donnée la plus ancienne. La deuxième forme de communication est dite synchrone et est la plupart du temps utilisée par les langages synchrones. Elle utilise la méthode du *rendez-vous* qui permet à deux tâches de se synchroniser à un instant précis de leur exécution pour s'exécuter par la suite de façon conjointe. Nous pouvons noter que la communication par rendez-vous peut être aisément simulée par deux boîtes aux lettres c'est pourquoi nous ne considérerons par la suite que ce type de communication.

gestion du temps.

Le temps est utilisé ici comme une horloge absolue pour cadencer le système. Nous utilisons traditionnellement une discrétisation du temps permettant au processeur d'effectuer une action atomique minimale au vu des instructions de l'application. La notion de temps intégrée dans un exécutif Temps Réel doit ainsi permettre de satisfaire plusieurs exigences [BW90] :

- l'accessibilité du temps courant pour permettre la mesure du temps écoulé.
- la mise en attente d'une tâche pendant une durée finie.
- la définition d'une minuterie ou *timer* pour la détection par exemple de la non occurrence d'un évènement attendu.

[1] Un ensemble de tâches est en situation d'interblocage si chacune attend un évènement que seule une autre tâche de l'ensemble peut engendrer. Typiquement, c'est le cas si chaque tâche attend une ressource détenue par l'une des autres tâches

– l'activation périodique de tâche.

Généralement, l'unité de temps est définie grâce à une source d'évènements périodiques faisant office d'horloge de base. Un quartz est utilisé afin d'assurer une plus grande régularité de la périodicité. Il convient toutefois de noter que la gestion du temps n'est pas sans conséquences. En effet, toutes les routines ou primitives temps réel de l'exécutif temps réel comme par exemple l'utilisation d'un timer pour spécifier un délai ou encore le choix de la tâche à activer, prennent un temps non négligeable au regard de l'horloge de base. En effet, le processeur doit non seulement exécuter les tâches mais également les instructions du système d'exploitation. Le pourcentage du temps processeur utilisé par le système d'exploitation est appelé *surcoût processeur* ou *overhead*.

gestion des interruptions et de la mémoire *etc. . .*.

La gestion des interruptions doit permettre de prendre en compte toutes les sollicitations matérielles et logicielles. Nous utilisons un service de routines d'interruption (ISR) permettant d'associer un traitement à chaque exécution. La durée de chaque routine doit être la plus courte possible puisque les routines s'exécutent de manière atomique (les interruptions sont masquées durant leurs exécutions).

La gestion de la mémoire peut être faite suivant deux modèles : soit l'exécutif et les tâches ont chacun une zone de mémoire réservée, soit chaque tâche ainsi que l'exécutif possèdent une zone mémoire séparée et protégée.

Toutes ces fonctions de l'exécutif Temps réel existent sous forme de primitives ou routines élémentaires dont la plupart possèdent des bribes atomiques, c'est à dire ne pouvant pas être interrompues par la gestion des interruptions matérielles. Ces portions ininterruptibles engendrent des retards dans la gestion des évènements qu'ils soient logiciels ou matériels. Pour assurer un service optimal aux traitements des tâches, il faut réduire ces portions au minimum. C'est justement l'un des critères d'évaluation des exécutifs Temps Réel du marché (ou RTOS pour Real Time Operating System), ce qui les différentient des systèmes d'exploitation classiques. Les RTOS assurent ainsi une borne temporelle pour chacune des primitives temps réel qu'elles proposent. Parmi ces RTOS, nous pouvons citer par exemple Osek/VDX, Vxworks, RTEMS, Lynx OS, Linux RT.

Il existe une autre fonctionnalité d'un exécutif Temps Réel, qui constitue le cœur de ce dernier, c'est l'*ordonnanceur*. En effet, le respect de l'échéance de chaque tâche dépend de l'instant où elle est exécutée par le processeur. Le rôle de l'ordonnanceur est de décider à tout instant quelle tâche doit être exécutée. Il décide donc de l'ordre d'exécution des tâches, ordre qui a une incidence forte sur le temps de réaction de l'application à un évènement. La règle qui régit le choix de l'ordre d'exécution constitue une *politique d'ordonnancement*. La détermination de cette politique est d'autant plus complexe que les tâches partagent des ressources ou nécessitent des synchronisations ou des communications. Ainsi, ces politiques d'ordonnancement doivent être définies de façon à tenir compte des problèmes engendrés par les exclusions mutuelles et l'utilisation de boites aux

lettres ou de rendez-vous, sans pour autant provoquer un manquement aux échéances des tâches.

1.3 Quantification du temps

L'architecture logicielle des applications temps réel permet d'identifier le traitement d'un évènement à une tâche. Nous avons vu que ce traitement doit intervenir dans des délais appropriés. Il faut donc être à même de vérifier que les contraintes temporelles sont bien respectées. Pour cela, nous devons introduire des indications temporelles quantitatives permettant par exemple d'exprimer les délais à respecter. Ceci est mis en œuvre par la modélisation temporelle des tâches. De plus, il est nécessaire de préciser la façon dont ces délais doivent être pris en compte. Nous devons, en d'autres termes, préciser la qualité de service attendue pour l'évaluation de l'application temps réel.

1.3.1 Les Tâches en Temps Réel

Il existe trois types de tâches en Temps Réel qui diffèrent par leurs caractéristiques temporelles. Les tâches dites *Périodiques* sont la plupart du temps stimulées par l'Horloge Temps Réel (HTR) de l'exécutif temps réel de façon à assurer une activité régulière, par exemple lors de l'acquisition de données (comme dans le cas d'une lecture échantillonnée d'un signal continu) ou la génération périodique d'évènements. Les *tâches apériodiques* sont quant à elles activées de façon aléatoire en fonction par exemple d'évènement aléatoire. Nous pouvons noter qu'il existe une sous famille de ce type de tâches qui est la famille des *tâche sporadiques* pour lesquelles une durée minimale sépare deux occurrences successives de l'évènement déclencheur. Enfin les *tâches cycliques*[HM95] sont très proches des tâches périodiques à la différence prêt que leur activation n'est pas liée à l'Horloge Temps Réel, ce qui induit une périodicité approximative. La durée séparant deux activations successives d'une tâche périodique est constante alors qu'elle appartient à un intervalle $[P_{min}, P_{max}]$ pour les tâches cycliques. Nous ne nous intéresserons par la suite qu'aux tâches périodiques et apériodiques.

Nous utiliserons le terme de tâche pour désigner le programme informatique compilé qui sera exécuté sur le processeur du système. Lorsque nous parlons d'activation d'une tâche, qu'elle soit périodique ou non, nous utiliserons le terme d'*instance* ou "*job*". Une instance de tâche désigne donc une exécution spécifique du code de la tâche concernée et il peut y avoir dans le système plusieurs instances de la même tâche. Nous donnons par la suite une description formelle de chaque type de tâche.

1.3.1.1 Tâche périodique

Le *modèle de tâche* périodique représente les tâches activées à intervalles réguliers (constants). Ce modèle s'appuie sur le *modèle temporel* initialement introduit par [LL73].

Définition 1 ***Modèle temporel d'une tâche périodique***

Soit une tâche périodique τ_i alors τ_i est modélisée par les quatre paramètres temporels :

$$\langle r_i, C_i, D_i, T_i \rangle$$

avec :
- *r_i la date à laquelle la première instance de τ_i est activée,*
- *C_i la pire durée d'exécution (ou charge maximale) de τ_i,*
- *D_i le délai critique (ou échéance relative) associé à τ_i ,*
- *T_i la période de la tâche τ_i.*

La figure 1.4 illustre le rôle de chacun de ces paramètres.

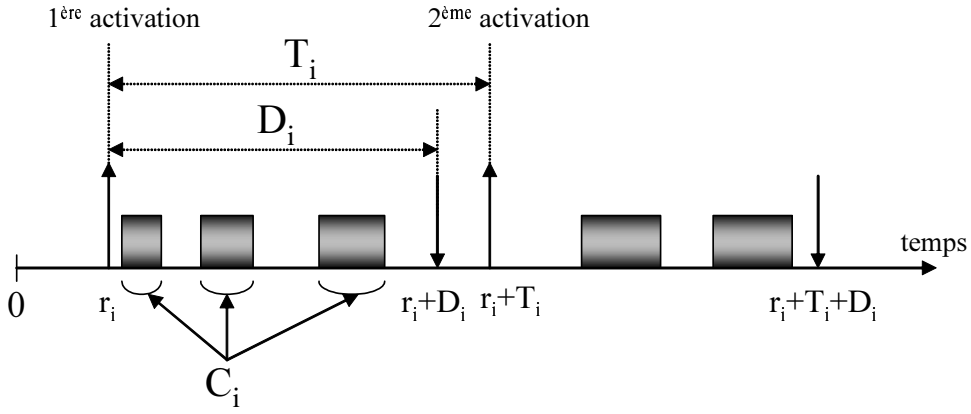

Fig. 1.4: Modélisation d'une tâche τ_i périodique.

Le date r_i de première instance de la tâche τ_i est appelée *date de réveil* ou "*offset*". Lorsque toutes les tâches ont la même date de réveil que nous pouvons alors supposer nulle ($r_1 = r_2 = \ldots = r_n = 0$), alors à l'instant $t = 0$, il existe une instance de chaque tâche prête à être exécutée. Nous parlons de tâches à départs *simultanés*. Dans le cas contraire nous parlons de tâches à départs *différés*. Lorsque la date de réveil n'est pas connue à l'avance, nous parlons de *tâche non concrète* ou "*Offset free*".

La pire durée d'exécution C_i est en réalité une borne supérieure du temps d'exécution de la tâche τ_i. Cette borne du pire cas d'exécution est très complexe à obtenir. En effet, que ce soit par analyse dynamique (mesure directe d'une exécution) ou analyse statique [PN98, KS86] (exploration du code de la tâche), son évaluation est rendue difficile par la présence d'instructions conditionnelles, de boucles indéterministes [PK89, GLB+02, HSRW98], ou encore des améliorations des processeurs comme l'exécution spéculative qui les rendent également indéterministes [MR02, RS02, CP00]. L'étude de ce paramètre appelé également *WCET* (Worst Case Execution Time) fait l'objet de nombreux travaux. Pour de plus

amples explications, les lecteurs peuvent se référer à [PB00, Col01, Pua05] ainsi qu'aux volumes 17 et 18 de la revue *Journal of Real-Time Systems* publiées respectivement en novembre 99 et mai 2000. Ce paramètre est par conséquent l'un des plus critiques pour la modélisation du système de tâches puisqu'il est le seul a priori à être une valeur approchée. C'est pourquoi, il existe de nombreux autres modèles de tâches permettant de tenir compte des fluctuations de cette durée d'exécution d'une instance de tâche à une autre. La section 1.3.1.3 présente quelques un de ces modèles.

Le délai critique D_i ou échéance relative, détermine le temps alloué à chaque instance de la tâche pour son exécution c'est à dire le délai maximal autorisé entre l'activation et la terminaison de l'instance. Ce paramètre est le plus intéressant du point de vue de la détermination de la correction et de l'efficacité du système.

Enfin, le dernier paramètre, T_i, quantifie l'intervalle de temps séparant deux activations successives d'une même tâche. Ainsi, il est possible de déterminer les dates d'activation pour chaque instance de la tâche τ_i : soit $k \in \mathbb{N}^+$ alors la date de réveil de la $k^{ième}$ instance de τ_i est

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

De la même façon, nous déterminons les dates successives des échéances de chaque instance d'une tâche τ_i avec $k \geq 1$ par

$$d_i^k = r_i^k + D_i$$

1.3.1.2 Tâche apériodique

Les tâches apériodiques ont pour origine des activations de deux types : elles peuvent provenir d'une intervention extérieure inattendue (comme une intervention humaine sur le procédé par exemple), ou provenir de l'application elle-même lorsque par exemple une tâche périodique chargée de faire de l'acquisition détecte une valeur inattendue nécessitant un traitement ponctuel spécifique. Leur importance dépend de la criticité de l'information qu'elles doivent traiter.

Les évènements déclencheurs étant en tout état de cause imprévisibles, les tâches apériodiques sont des tâches dont la fréquence d'activation est totalement aléatoire. Les paramètres du modèle précédent comme les dates de réveil et périodes n'ont par conséquent plus lieu d'être ici. Par contre, une tâche apériodique possède bien une durée d'exécution bornée par un WCET C_i et éventuellement un délai critique D_i pour s'assurer de son exécution dans un temps borné.

Les tâches sporadiques possèdent un paramètre supplémentaire permettant de définir un intervalle minimal entre deux activations successives [Mok83]. Cet intervalle minimal est généralement assimilé à son délai critique. Il existe de nombreuses définitions et particularités sur ce type de tâches. Pour une plus grande homogénéité des définitions, nous utilisons la terminologie de [But97]. Ainsi toute tâche apériodique munie d'un délai cri-

tique sera considérée comme une tâche sporadique. Cette considération permet d'éviter les problèmes de *réentrance* que nous verrons à la section suivante.

Définition 2 ***Modèle temporel des tâches apériodiques et sporadiques***

Soit une tâche τ_i.
- *Si τ_i est* ***une tâche apériodique****, alors τ_i est modélisée par un unique paramètre temporel : C_i sa pire durée d'exécution.*
- *Si τ_i est une* ***tâche sporadique****, alors τ_i est modélisée dans le cas général par :*

$$\langle C_i, D_i, Ts_i \rangle$$

où Ts_i correspond à l'intervalle de temps minimum séparant deux activations successives et D_i au délai critique. Nous considérons par la suite que $D_i = Ts_i$.

Notons que nous pouvons unifier ces modèles de tâche en considérant les *tâches événementielles* [KRP+93] qui généralisent les paramètres temporels en se basant sur les évènements qui activent chaque instance de tâches. Ainsi, en définissant les lois d'arrivées de ces activations (pouvant provenir d'interruptions matérielles ou logicielles) il est possible de redéfinir les tâches périodiques et apériodiques suivant la régularité de ces évènements. Par la suite, nous n'utiliserons pas ce dernier modèle.

1.3.1.3 Modèles de tâche enrichis

Nous avons vu précédemment que le paramètre C_i est difficile à évaluer. Il s'en suit que les valeurs produites sont souvent surévaluées (cette surestimation est de l'ordre de 5 à 9%). Lorsque le système est très chargé, c'est à dire lorsque les tâches occupent fortement le processeur, cette surestimation peut nuire à la garantie du respect des délais et conduire à juger de mauvaise qualité une application qui ne l'est pas nécessairement. Pour palier ce problème divers modèles temporels de tâches ont vu le jour. Ils permettent de réduire ce pessimisme, mais leur analyse temporelle est en contrepartie plus complexe. Ils contournent le principe même du calcul de la pire durée d'exécution d'une tâche. Un tel calcul est justifié par le fait que d'une instance à l'autre d'une même tâche, les contextes d'exécution peuvent être différents (variables d'entrée de la tâche et état du système). Par conséquent, le comportement d'exécution de la tâche sera différent d'une instance à l'autre. Les modèles de tâches que nous présentons ci-après sont dit enrichis en ce qu'ils utilisent non pas une seule valeur de durée d'exécution mais plusieurs, chacune étant liée à un comportement spécifique d'exécution de la tâche.

Les premiers modèles enrichis ont été introduits pour traiter la problématique de la tolérance aux fautes. Parmi ceux-si citons :

Le modèle à multi représentations [CHB79, CC91]
qui associe à une tâche deux implémentations donc deux durées d'exécution, l'une

correspondant à une qualité de service optimale, mais de durée incertaine, et l'autre, de durée connue, mais correspondant à un service de moindre qualité.

Le modèle multiframe [MC96]
permet d'avoir des temps d'exécution qui suivent un motif donné. Nous supposons donc que nous connaissons à l'avance les durées d'exécutions des différentes instances d'une même tâche. Mais ce premier modèle ne tient pas compte des échéances explicitement, il faut ainsi attendre la généralisation des multiframes (Generalize Multiframe Model) par [BCGM99], pour voir apparaître des échéances relatives et périodes pouvant elles même être variables suivant un motif.

Le modèle de tâche récurrent (recurring tasks) [Bar98, Bar03]
permet de découper chaque tâche en un nombre fini de sous-tâches avec échéances relatives internes et des précédences. Cette représentation des tâches autorise ainsi la prise en compte des instructions conditionnelles des tâches. Chaque tâche est représenté par un graphe orienté acyclique composé d'un noeud source représentant la première sous-tâche de la tâche et un unique noeud feuille décrivant la dernière sous-tâche. Chaque nœud représente donc une sous-tâche munie d'une durée d'exécution et d'une échéance propre et chaque lien est un possible flot de contrôle étiqueté par une durée minimale entre le moment où l'origine du lien est activée et l'activation du noeud pointé par le lien. Cela permet donc de reconstruire grâce aux sous-tâches des flots de contrôle comme des instructions conditionnelles. Ce modèle peut être considéré comme une extension naturelle du modèle multiframe.

1.3.1.4 Mesures associées

Les modèles de tâches possédant un paramètre de délai critique D_i, peuvent être classifiés en trois catégories suivant les relations existant entre D_i et T_i :

- $\mathcal{A} = \{$ *Tâche à échéance non reliée à la période*$\}$. Aucun relation n'existe entre D_i et T_i donc toutes les cas sont possibles. Il faut toutefois noter que dans le cas où $D_i > T_i$, une instance de tâche peut être activée alors que la précédente n'a toujours pas terminé son exécution. C'est ce que nous appellons le phénomène de *réentrance*.
- $\mathcal{B} = \{$ *Tâche à échéance inférieure à la période*$\}$. Toutes les tâches vérifiant la relation $D_i \leq T_i$ rentre dans cette catégorie.
- $\mathcal{C} = \{$ *Tâche à échéance égale à la période*$\}$. Cette catégorie regroupe toutes les tâches vérifiant la relation $D_i = T_i$. Nous parlons alors de tâches à *échéance sur requête*. Nous souhaitons ici simplement interdire la réentrance. Notons que cette catégorie est certainement la classe la plus utilisée.

Nous pouvons remarquer que $\mathcal{C} \subset \mathcal{B} \subset \mathcal{A}$. La définition des modèles de tâches permet de définir des critères mesurable pour la détermination de la correction et l'efficacité de l'exécution des applications temps réel. Ces critères vont permettre entre autre de décider de la politique d'ordonnancement à intégrer dans l'ordonnanceur de l'exécutif temps réel suivant les contraintes techniques voulues par les concepteurs du système temps réel. La définition 3 énumère les critères les plus couramment utilisés.

Définition 3 ***Mesures liées à l'exécution d'une application temps réel***

Soit une application temps réel composée de n tâches. Nous définissons le systèmes de tâches par $\mathcal{S} = (\tau_1, \ldots, \tau_n)$. *Soit* $k \in \mathbb{N}$, *alors pour une tâche* $\tau_{i_{1 \leq i \leq n}}$, τ_i^k *dénote sa* $k^{ème}$ *instance. Nous définissons les critères suivant :*

- $d(i,k)$ *est la date absolue pour laquelle* τ_i^k *exécute sa première instruction.*
- $f(i,k)$ *est la date absolue à laquelle* τ_i^k *termine sa dernière instruction.*
- $C_i(t)$ *est la* ***durée d'exécution dynamique****. Il s'agit du temps restant au processeur pour terminer l'exécution de l'instance en cours de la tâche* τ_i *à la date t.*
- u_i *correspond au* ***facteur de charge processeur*** *occasionné par la tâche* τ_i. *Il s'agit de la proportion de temps processeur que* τ_i *va monopoliser pour s'exécuter. Cette quantité est évaluée par* $u_i = \frac{C_i}{T_i}$.
- U *correspond à la* ***charge processeur*** *induite par toutes les tâches de* $\mathcal{S}$, $U = \sum_1^n u_i$
- TR_i^k *détermine le* ***temps de réponse*** *de* τ_i^k. *Le temps de réponse se calcule par* $TR_i^k = f(i,k) - r_i^k$
- TR_i *définit le* ***pire temps de réponse*** *ou Worst Case Response Time de* τ_i *pour toutes ses instances. Nous avons donc* $TR_i = max_{k \in \mathbb{N}^+} \left\{ TR_i^k \right\}$
- L_i *correspond à la* ***laxité initiale*** *de la tâche* τ_i. *Cette laxité correspond à la durée maximale dont une instance de* τ_i *peut retarder son exécution sans dépasser son échéance. Nous évaluons* L_i *par* $L_i = D_i - C_i$
- $L_i(t)$ *est la* ***laxité dynamique*** *de* τ_i. *Elle reprend la définition associé à* L_i *mais en ne considérant que la durée d'exécution restante* $C_i(t)$ *soit* $L_i(t) = d_i^k - t - C_i(t)$.
- Lat_i^k *définit le temps de retard admissible ou* ***latence***. Lat_i^k *est défini par* $Lat_i^k = d_i^k - f(i,k)$
- Jit_i *définit* ***la gigue temporelle*** *ou* ***jitter****, qui permet de caractériser la régularité d'exécution de toutes les instances de* τ_i. *Bien entendu ce critère n'a de sens que pour les tâches périodiques. Nous évaluons cette gigue temporelle en fonction des temps de réponse de toutes les instances successives de* τ_i, *ainsi* Jit_i *est défini par* $Jit_i = max_{k \in \mathbb{N}^+} \left\{ \frac{\left| TR_i^{k+1} - TR_i^k \right|}{D_i} \right\}$

La figure 1.5 illustre pour une tâche τ_i ces différentes mesures.

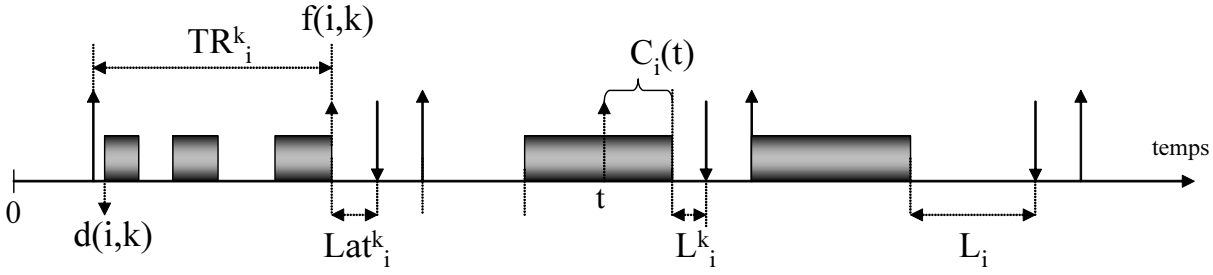

Fig. 1.5: Critères d'évaluation de l'exécution d'une tâche

1.3.2 Qualité de service

Dans le paragraphe précédent, nous avons défini les paramètres temporels caractérisant une tâche. Il nous faut maintenant indiquer la nature des contraintes qu'ils engendrent. En effet, les systèmes temps Réel n'ont pas tous le même degré d'exigence vis vis de ces critères. Si nous considérons un système critique embarqué dans un avion, il est vital que les tâches d'un tel système aient des temps réponse rigoureusement contrôlés, inférieurs systématiquement à une borne fixée (exprimée par le délai critique des tâches). Au contraire, un retard de réaction (par rapport aux bornes fixées par les concepteurs qui correspondent à un fonctionnement optimal) lors de la compression vidéo [HST97, IFS03] n'entraîne aucune catastrophe, ni même de perturbation sensible si ce retard n'intervient pas trop souvent. Cette constatation permet de définir des classes de systèmes temps réel suivant la degré de criticité de leur qualité de service. On distingue ainsi 3 familles de systèmes temps réel suivant la rigidité des contraintes temporelles qui leurs sont imposées. Celles-ci sont définies dans la définition 4.

Définition 4 ***Contextes temporels des systèmes temps réel***

Les systèmes temps réel se répartissent suivant trois domaines d'applications liés à la criticité des contraintes temporelles. Nous distinguons :

- *Les Systèmes Temps Réel à* ***Contraintes Strictes****. Ce type de système impose que toutes les contraintes temporelles soient impérativement respectées et plus particulièrement* $f(i,k) \leq d_i^k$.
- *Les Systèmes Temps Réel à* ***Contraintes Souples****. À l'opposé de la classe précédente, un non respect d'une échéance n'entraîne pas la défaillance du système. Ces dépassement sont donc tolérés mais entraîne des perturbations qu'il faudra alors minimiser.*
- *Les Systèmes Temps Réel à* ***Contraintes Mixtes****. Ces derniers sont soumis à la fois aux exigences des systèmes à contraintes strictes pour certaines tâches et à celles des systèmes à contraintes souples pour d'autres.*

Les Systèmes Temps Réel à Contraintes Strictes ont un comportement déterministe. Nous pouvons les rencontrer dans des systèmes du domaine de l'embarqué comme l'avionique, la robotique, les systèmes ABS *etc.*... Les Systèmes Temps Réel à Contraintes Souples dont la mesure d'efficacité s'opère généralement par une analyse statistique des temps de réponse moyen peuvent se rencontrer dans les systèmes du traitement multimédia comme le "streaming" par exemple. Enfin, les Systèmes Temps Réel à Contraintes Mixtes regroupe des applications temps réel composées de tâches dont un sous ensemble doit impérativement respecter des contraintes temporelles, à l'inverse des autres tâches dont le critère d'évaluation cherchera à minimiser les fautes temporelles. Ces systèmes regroupe la plupart des systèmes temps réel actuels.

Il existe d'autres classifications plus spécifiques qui semblent être de plus en plus utilisées. Par exemple, nous pouvons affiner la classe des Systèmes Temps Réel à contraintes Souples en introduisant la notion de *Systèmes Temps Réel ferme* ("Firm real Time system") [HR95]. En effet, certaines applications ayant des contraintes temps réel souples doivent respecter un temps de réponse moyen tout en restant à l'intérieur d'une fenêtre tempo-

relle stricte. Suivant que le non-respect d'une contrainte temporelle continue d'apporter quelque chose au système ou non : si le traitement dépasse une contrainte temporelle et si continuer son exécution dans ces conditions jusqu'à son terme n'apporte rien au système, nous parlons de traitement temps-réel ferme.

Certains Systèmes Temps Réel Stricts ou Souples peuvent posséder des critères d'efficacité comme le temps d'exécution passé sur une tâche en particulier. Dans ce cas le modèle de tâche ne comporte plus de durée d'exécution fixe mais peut être séparé en une partie obligatoire devant respecter une échéance stricte et une partie incrémentale à résultat imprécis sans échéance [CL88]. Nous appelons de tels systèmes les *Systèmes Temps Réel Incrémentaux*. La qualité de la réponse des applications à contraintes de temps incrémentales s'accroît avec le temps de calcul de la partie incrémentale. Ainsi, nous pouvons juger de l'efficacité d'une telle application si à l'échéance du traitement D_i, la qualité du résultat fourni est suffisant. Ces applications sont utilisés à leurs limites pour diminuer le temps au bout duquel la qualité est atteinte ou bien pour obtenir la meilleure qualité à temps de réponse constant. De tels systèmes se trouvent par exemple en traitement d'images (reconnaissance d'objets), dans les applications multimédia (compression vidéo) ou en calcul scientifique.

La figure 1.6 résume ces différentes classes de systèmes.

Par la suite nous ne nous intéresserons qu'aux Systèmes Temps Réel à Contraintes Strictes.

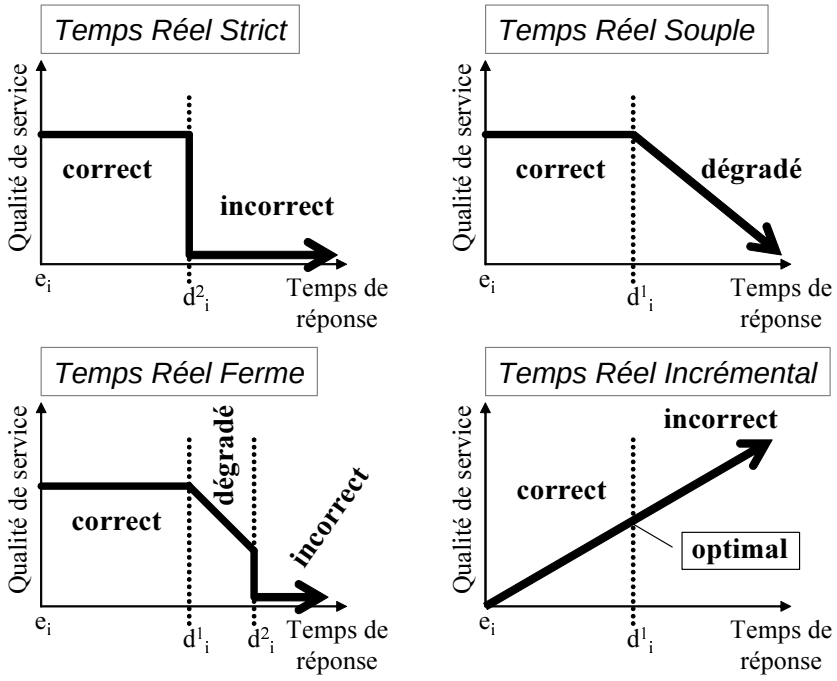

Fig. 1.6: Quatre types de contraintes Temps Réel offrant des qualités de service différentes. Les Systèmes Temps Réel à contraintes mixtes héritent à la fois de tâches à contrainte strictes et de tâches à contraintes souples.

1.4 Bibliographie

[AD92] M. Alabau and T. Dechaize. Ordonnancement temps réel par échéance. In *T.S.I.*, volume 11. n.3, 1992.

[Bar98] S. Baruah. A general model for recurring real-time tasks. In *Proceeding of the Real Time System Symposium, IEEE Computer Society Press*, pages 114–122, Madrid, Spain, 1998.

[Bar03] Sanjoy K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Syst.*, 24(1) :93–128, 2003.

[BCGM99] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized multiframe tasks. *Real-Time Syst.*, 17(1) :5–22, 1999.

[BG92] Gerard Berry and Georges Gonthier. The esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[But97] G.C. Buttazzo. *Hard real time computing systems : predictable scheduling algorithms and applications*. Kluwer Academic Publisher, 1997.

[BW90] A. Burns and A. Wellings. *Real-Time Systems and their Programming Languages*. Addison-Wesley, 1990.

[CC91] H. Chetto and M. Chetto. An adaptative scheduling algorithm for fault-tolerant real-time systems. In *Software Engineering Journal*, pages 179–186, Mai 1991.

[CDKM00] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Oronnancement Temps Réel*. Hermès Edition, 2000.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.

[CHB79] R.H. Campbell, K.H. Hurton, and G.G. Belford. Simulations of fault-tolerant real-time deadlin mechanisms. In *Proceedings of FCTS'9*, pages 95–101, Janvier 1979.

[CL88] J.Y. Chung and J.W.S Liu. Algorithms for scheduling periodics jobs to minimize average error. In *9th IEEE RTSS*, pages 142–152, Décembre 1988.

[CNR88] G.D.R.T.R CNRS. Le temps réel. *Technique et Science Informatiques*, 1988.

[Col01] A. Colin. *Estimation de temps d'exécution au pire cas par analyse statique et application aux systèmes d'exploitation temps-réel*. PhD thesis, niversité de Rennes I, October 2001.

[CP00] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2) :249–274, May 2000.

[Ell97] Jean Pierre Elloy. Systèmes réactifs synchrones et asynchrones. In *Applications, Réseaux et Systèmes – École d'été temps réel'99*, pages 43–51, Futuroscope, Septembre 1997.

[GLB+02] J. Gustafsson, B. Lisper, N. Bermudo, C. Sandberg, and L. Sjöberg. A prototype tool for flow analysis of c program. In *2nd Intl. Workshop on WCET Analysis 14th Euromicro Conference on Real- Time Systems*, June 2002.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[HM95] C. Hanen and A. Munier. *Cyclic scheduling on parallel processors : An Overview*, volume Scheduling theory and its applications, P. Chretienne et al., Chap 9. John Wiley & Sons, 1995.

[HR95] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with, 1995.

[HSRW98] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, Discrete Event Systems : Analysis and Control, pages 12–21. IEEE, 1998.

[HST97] H. Hansson, M. Sjodin, and K. Tindell. Guaranteeing real-time traffic through an atm network. *In Proc. of the 30'th Hawaii International Conference on System Sciences, IEEE Computer Society Press*, 5 :44–53., January 1997.

[IFS03] D. Isovic, G. Fohler, and L. Steffens. Timing constraints of mpeg-2 decoding for high quality video : misconceptions and realistic assumptions. *In Proceedings of the 15th Euromicro Conference on Real-Time Systems*, June 2003.

[IFS04] Damir Isovic, Gerhard Fohler, and Liesbeth Steffens. Real-time issues of mpeg-2 playout in resource constrained systems. *International Journal on Embedded Systems*, 1, issue 2(ISSN 1740-4460), 6 2004.

[Kai01] C. Kaiser. *Description et critique d un système temps réel pour le suivi d un laminoir : Robustesse et potentiel d'évolutivité*, volume 20(1). Hermes Science, 1901.

[Kra85] S. Krakowiak. *Principes des systèmes d'exploitation.* Dunod, 1985.

[KRP+93] M. Klein, T. Rayla, B. Pollak, R. Obenza, and M.G. Harbour. *A practitionner's handbook for real time systems analysis.* Kluwer Academic Publisher, 1993.

[KS86] E. Klingerman and A.D. Stoyenko. Real time euclid : a language for reliable real time systems. *IEEE Trans. softw. Eng.*, 12(9) :941–949, 1986.

[KS99] C. Kaiser and G. Stoffel. Systeme d acquisition et d analyse en temps reel des signaux d un laminoir. *Rapport scientifique CEDRIC*, 1999.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for mutltiprogramming in real-time environnement. *Journal of the ACM*, 20(1) :46–61, 1973.

[MC96] A. K. Mok and D. Chen. A multiframe model for real-time tasks. In *RTSS '96 : Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 22, Washington, DC, USA, 1996. IEEE Computer Society.

[Mok83] A.K. Mok. *Fundamental design problems for the hard real time environments.* PhD thesis, MIT, 1983.

[MR02] T. Mitra and A. Roychoudhury. A framework to model branch prediction for wcet analysis. In *2nd Intl. Workshop on WCET Analysis 14th Euromicro Conference on Real- Time Systems*, June 2002.

[PB00] P. Puschner and A. Burns. Guest editorial : A review of worst-case execution-time analysis. *Real-Time Systems*, 18 :115–128, 2000.

[PK89] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2) :159–176, 1989.

[PN98] P. Puschner and R. Nossal. Testing the results of static worst case execution time analysis. In *IEEE Real Time Systems Symposium*, dec 1998.

[Pua05] I. Puaut. Méthodes de calcul de wcet (worst case execution time) etat de l'art. *Ecole d'été Temps Réel*, 4 :165–175, Septembre 2005.

[RCK01] Pascal Richard, Francis Cottet, and Claude Kaiser. Précédences généralisées et ordonnançabilité des tâches de suivi temps réel d'un laminoir. *Journal Européen des Systèmes Automatisés*, 35 (9) :1055–1071, 2001.

[Rou99] O. Roux. Les langages reactifs synchrones et asynchrones. *Chapitre des Techniques de l'Ingenieur, Traite Controle et Mesure*, 11,(4) :448–471, december 1999.

[RS02] C. Rochange and P. Sainrat. Difficulties in computing the wcet for processors with speculative execution. In *2nd Intl. Workshop on WCET Analysis 14th Euromicro Conference on Real- Time Systems*, June 2002.

[Sta88] J.A. Stankovic. Misconception about real -time computing. In *IEEE Computer Magazine*, volume 10, pages 0–19. 21, 1988.

Chapitre Deux

Ordonnancement et Validation

Chapitre Deux

Ordonnancement et Validation

Dans cette deuxième partie nous établissons un état de l'art des techniques d'ordonnancement temps réel. Nous présentons plus particulièrement les problématiques de l'ordonnancement et en donnons quelques définitions. Nous proposons par la suite un bref descriptif des différentes classes de problème. Enfin, nous nous penchons sur les deux grands courant d'ordonnancement temps réel que sont les approches "en ligne" et "hors-ligne". Nous exposons plus particulièrement leurs avantages et inconvénients respectifs notamment en présence de ressources.

Sommaire

Chapitre Deux

Ordonnancement et Validation

Nous venons de définir les systèmes temps réel. Nous avons mis en évidence les différents éléments qui les composent : capteurs/actionneurs, exécutif temps réel, application multitâches, ainsi que la notion de programmation concurrente permettant l'exécution en pseudo-parallélisme (dans le cas monoprocesseur) des différentes tâches composant l'application. Or, l'ordre de ces exécutions a une importance majeure au regard des contraintes temporelles liées à chaque tâche. En effet, prenons l'exemple d'une application composée de 5 tâches, celle-ci peut être exécutée sur un processeur de 120 façons différentes si nous interdisons la préemption. Or, si le résultat de l'application est identique quelque soit la façon dont elle a été exécutée, le comportement temporel va varier considérablement ce qui peut empêcher certaines tâches de respecter leurs contraintes temporelles. Un système temps réel doit donc faire en sorte que les contraintes temporelles soient respectées durant l'exécution concurrente de l'application. Nous utilisons pour cela la théorie de l'ordonnancement et la validation qui permettent de s'assurer du bon comportement temporel de l'application concurrente durant son exécution.

2.1 Problématique et Définition

Un des composants fondamentaux d'un système temps réel est l'*ordonnanceur*. Celui-ci est chargé d'élire à tout instant la tâche à exécuter. Chaque élection d'une nouvelle tâche s'accompagne d'un changement de contexte, permettant de passer de l'exécution de l'ancienne tâche à la nouvelle. Cette succession d'élection permet de définir une *séquence d'ordonnancement* qui est constituée d'une suite $(\tau_{i_0}, \tau_{i_1}, \ldots)$ telle que τ_{i_k} est la tâche s'exécutant à l'instant k. Une telle séquence est une application :

$$\begin{array}{rcl} \mathbb{N} & \longmapsto & (\tau_1, \ldots, \tau_n) \\ k & \longmapsto & \tau_{i_k} \end{array}$$

qui peut être visualisée par un diagramme de Gantt (cf. figure 2.1).

La problématique de l'ordonnancement consiste à définir une politique d'élection adéquate, c'est à dire garantissant le respect des contraintes temporelles. Cette politique d'élection peut s'exprimer par une simple politique d'ordonnancement basée sur des priorités ou par un algorithme plus complexe et doit permettre de définir une séquence d'ordonnancement valide.

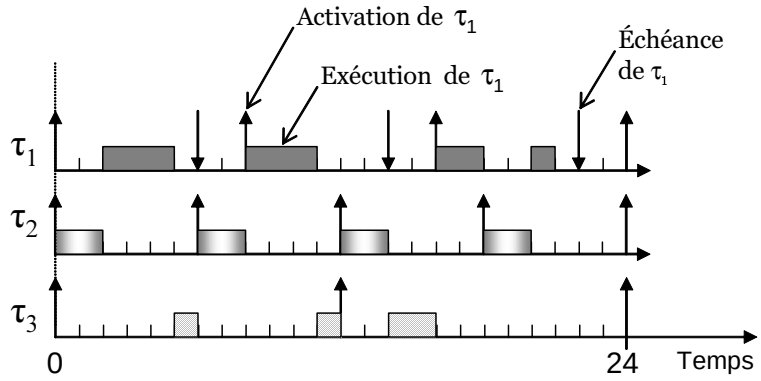

Fig. 2.1: Diagramme de Gantt représentant une séquence d'ordonnancement d'une configuration de 3 tâches $\tau_1 \langle 0, 3, 6, 8 \rangle$, $\tau_2 \langle 0, 2, 6, 6 \rangle$, $\tau_3 \langle 0, 2, 12, 12 \rangle$

Définition 5 ***Ordonnancement valide***

Une séquence d'ordonnancement est ***valide*** *si toutes les contraintes temporelles, de ressources et de précédences sont respectées.*

Soit une *configuration de tâches* définie par une application temps réel multitâches composée de n tâches notées τ_i pour $1 \leq i \leq n$ utilisant r ressources partagées et c communications. Le *problème de la validation* d'une configuration de tâche consiste à déterminer s'il existe une solution au problème de l'ordonnancement. Deux approches sont envisageables à partir d'une configuration de tâches :

- Répondre à la question de l'ordonnançabilité de la configuration de tâches. Il s'agit de savoir s'il existe ou non une séquence d'ordonnancement valide pour cette configuration. Si tel est le cas, nous pouvons également chercher à exhiber une solution.
- Déterminer si pour cette configuration de tâche et pour une politique ou algorithme d'ordonnancement donnée, la séquence d'ordonnancement produite est valide. Si c'est effectivement le cas, nous pouvons dire que l'algorithme d'ordonnancement est valide.

Nous pouvons noter que pour une configuration de tâche ordonnançable, il peut exister de nombreux algorithmes d'ordonnancement valides. L'un des problèmes centraux lorsque nous souhaitons valider une application réside donc dans le choix de la stratégie d'ordonnancement à mettre en œuvre. Ce choix peut être simplifié pour certaines familles d'applications pour lesquelles il existe des algorithmes d'ordonnancement dits optimaux.

Définition 6 ***Algorithme d'Ordonnancement Optimal***

Soit **T** *une classe d'applications. Un algorithme d'ordonnancement est dit* ***optimal*** *(éventuellement parmi une sous-famille d'ordonnancements) pour les applications de la classe* **T** *si et seulement si quelle que soit l'application de la classe, soit l'algorithme l'ordonnance de manière correcte, soit aucun autre algorithme (de la sous-famille) ne le pourra.*

Une séquence d'ordonnancement peut être produite suivant deux approches différentes en fonction des critères temporels qui ont été définis à partir de l'application temps réel. Soit ils sont suffisamment nombreux et connus a priori, dans ce cas il est possible d'élaborer un algorithme qui en tire profit, nous parlons ainsi d'ordonnancement *statique* puisqu'il est possible de planifier à l'avance la séquence d'ordonnancement. Soit, l'algorithme ne s'occupe que des tâches actives à chaque instant pour prendre une décision d'ordonnancement et nous parlons alors d'ordonnancement *dynamique*.

Lorsque la charge processeur est inférieure à 1, c'est à dire $U < 1$, ce qui arrive dans la plupart des systèmes temps réel, le processeur n'est pas occupé à plein temps. Il y a donc des temps d'inactivité appelé *temps creux*. Généralement ces temps creux ne sont pas gérés par l'algorithme d'ordonnancement, mais nous verrons ultérieurement que certaines politiques permettent de les placer volontairement à certains instant (en anticipant leurs venue). Nous pouvons ainsi caractériser les algorithmes d'ordonnancement qui tiennent compte ou non de ces temps creux.

Définition 7 ***Ordonnancement conservatif ou au plus tôt***

Un algorithme d'ordonnancement est dit ***conservatif*** *ou au plus tôt, lorsqu'il ne prend jamais la décision de ne rien faire lorsqu'au moins une tâche est active et non bloquée.*

Il est souvent utile pour faciliter la validation d'un algorithme d'ordonnancement, de faire en sorte que ce dernier ait un comportement identique lorsqu'il est confronté à l'exécution d'une même configuration de tâches actives. Nous parlons alors d'algorithme déterministe.

Définition 8 ***Algorithme d'Ordonnancement Déterministe***

Un algorithme d'ordonnancement est dit ***déterministe*** *s'il prend toujours la même décision devant une configuration de tâche identique.*

Nous venons de caractériser les algorithmes d'ordonnancement implantés dans l'ordonnanceur des système temps réel. Dans le cas des Systèmes Temps Réel à Contraintes Strictes, la fiabilité du système doit impérativement être *validée temporellement*. Que nous voulions répondre au problème de l'ordonnançabilité ou vérifier la validité d'une séquence produite par un algorithme d'ordonnancement, il est nécessaire de prouver que quelque soit la situation dans laquelle se trouve le système (dans un cadre normal d'utilisation)l'exécution de l'application multi-tâches respectera ses contraintes temporelles.

Cette étude de validation temporelle repose sur deux approches :

- soit *analytique* : il s'agit de de s'assurer qu'un critère analytique utilisant les paramètres temporels des tâches est vérifié.
- soit par *simulation* : il s'agit de simuler le fonctionnement du système et s'assurer qu'il est correct. Il devient alors indispensable de restreindre la simulation à une durée d'étude bornée en temps.

Cette dernière approche est utilisée lorsqu'aucun résultat probant ne peut être fourni par la méthode analytique.

Pour répondre au problème de la validation temporelle, il est nécessaire préalablement de savoir s'il existe une solution au problème de l'ordonnancement. L'obtention d'un ordonnancement peut s'effectuer de deux façons différentes :

- l'ordonnancement *en ligne* : nous implémentons un algorithme au niveau de l'ordonnanceur, les décisions d'ordonnancement étant prises au cours de la vie de l'application.
- l'ordonnancement *hors ligne* : une séquence valide est calculée avant l'exécution effective de l'application. Elle est ensuite chargée dans une table utilisée par le séquenceur au cours de la vie de l'application.

L'utilisation d'un ordonnancement hors ligne permet d'éviter les surcharges processeur liées à l'exécution d'un algorithme d'ordonnancement et présente davantage de garanties en cas d'utilisation de ressources critiques. En contrepartie, un ordonnancement en ligne est plus souple, en particulier en cas de reconfiguration de l'application, ou en cas de prise en compte de tâches sporadiques.

2.1.1 Approches En-Ligne

Cette approche utilise un algorithme qui est directement implanté et exécuté par l'ordonnanceur. À chaque exécution de primitives temps réel (prise de ressource, préemption *etc...*) l'ordonnanceur intervient et choisit la tâche à affecter au processeur. Ce choix est par conséquent dynamique et dépend le plus souvent des paramètres des tâches présentes dans la file des tâches actives à l'instant t. Nous appelons ce type d'ordonnancement, *l'ordonnancement en-ligne*.

Les algorithmes utilisés reposent le plus souvent sur la notion de *priorité* permettant la construction de plan d'ordonnancement. Les priorités sont déterminées soit en cours de fonctionnement, auquel cas nous parlons d'*algorithme à priorités dynamiques* [KSSK96], soit, plus fréquemment, avant le fonctionnement, nous parlons alors d'*algorithme à priorités fixes* ou *statiques*. Pour bien distinguer leurs caractéristiques respectives, il suffit de constater que dans le cas des priorités fixes, les priorités sont attribuées aux tâches alors que dans le cas des priorité dynamiques, les propriétés sont attribuées aux instances de tâches.

La validation des algorithmes en-ligne repose, soit sur un *test d'ordonnançabilité* basé sur les critères temporels des tâches, soit sur une simulation d'exécution durant une durée suffisante (cf. 2.4). Si ce test ne permet pas d'aboutir à une validation par exemple dans les cas très contraints (présence de ressources et de synchronisations), seule une simulation permet de se rendre compte du bon comportement de l'application.

L'ordonnancement en ligne bénéficie d'une grande souplesse d'implémentation ainsi, l'ajout d'une nouvelle tâche dans le système peut être aisément effectué tant que l'algorithme d'ordonnancement reste valide. De plus, dans certain cas, il est possible d'obtenir des critères analytiques d'ordonnançabilité des algorithmes ou encore utiliser des algorithmes optimaux suivant la configuration des tâches. En revanche, cette approche est sujette à quelques inconvénients. En effet, les critères analytiques utilisés par le test d'ordonnançabilité sont soit des conditions nécessaire et suffisantes soit des conditions

suffisantes, dans le contexte précis utilisé par l'algorithme d'ordonnancement. Nous notons que ces critères analytiques reposent sur un modèle temporel de tâches et que ce dernier ne reflète pas à 100% l'exact comportement d'exécution des tâches ce qui nuit à la puissance d'ordonnancement de cette approche. De plus, il est courant que la durée d'exécution d'une tâche utilisée dans le modèle temporel représente la pire durée d'exécution possible qui n'est que très rarement atteint en réalité. Il se pose donc des problèmes d'instabilité du système mis en évidence en 2.3.4 malgré un test d'ordonnançabilité certifiant la validation. Par conséquent, si cette approche permet une grande flexibilité quant à la modification en ligne de l'application (tant que le test d'ordonançbilité reste vrai) celle ci souffre d'un trop grand pessismisme quant aux possibilités d'ordonnancement dans les cas des conditions suffisantes.

2.1.2 Approches Hors-ligne

Il s'agit ici de calculer à l'avance la séquence des tâches au fil du temps (dates de début d'exécution, de préemption, de reprise d'exécution). La *séquence d'ordonnancement* ou plan hors-ligne ainsi produite, est exécutée de façon répétitive (ou cyclique). Cette séquence est placée dans une table qui est utilisée par le séquenceur pour affecter le processeur aux tâches. Nous appelons cette approche l'*ordonnancement Hors-ligne* et nous pouvons noter que le nom de *Séquenceur* est alors préférable à celui d'ordonnanceur.

L'utilisation de cette méthode se justifie dès lors que l'on souhaite ordonnancer des applications fortement couplées (forte utilisation de ressources et de synchronisations). En effet, la construction d'une séquence d'ordonnancement permet de s'assurer du respect des contraintes temporelles pendant sa construction et donc de valider du même coup la séquence produite puisque celle-ci sera utilisée telle quelle dans le séquenceur. Par contre, ce calcul préalable nécessite de connaître parfaitement toutes les tâches présentes dans l'application ainsi que leurs dates d'occurrences, ce qui fige à jamais l'application. Il convient de construire une autre séquence dès lors qu'une modification de l'application est opérée comme un ajout d'une tâche ou la modification de l'une d'entre elles entraînant une modification de ses paramètres temporels.

Les méthodes de constructions de séquences d'ordonnancement reposent généralement sur une analyse exhaustive de toutes les séquences possibles pour en extraire une particulière valide du point de vue temporel. Par conséquent, ces méthodes souffrent d'une complexité élevée (cf. complexité des différentes classes de problèmes 2.2).

En contre partie, l'étude exhaustive permet de parcourir l'ensemble des solutions possibles ce qui permet d'offrir une puissance d'ordonnancement maximale. De plus, le fait de s'absoudre d'un ordonnanceur, remplacé par un séquenceur, permet d'alléger le processeur en surcoût d'ordonnancement, tout en fiabilisant l'exécution de l'application qui n'est alors plus assujettie aux phénomènes d'instabilité (2.3.4).

2.2 Classe des Problèmes

Avant d'énumérer les différents résultats sur l'ordonnançabilité des applications temps réel, il convient de différencier les classes de problème. Définir un ordonnancement peut être ramené à un problème d'optimisation. Par exemple, beaucoup de travaux s'intéressent

à minimiser le retard maximal parmi toutes les tâches ($Lat_{max} = max_{i=1,...,n;k\in\mathbb{N}}(Lat_i^k)$). Une fois ce retard maximisé, prouver la correction temporelle du système est équivalent à vérifier que ce retard maximal est négatif ou nul. L'ordonnançabilité est donc un problème de décision et l'ordonnancement correspond à la résolution d'un problème d'optimisation. Ces deux problèmes se traduisent en théorie de l'ordonnancement par [BG04, Che02] :

– un test d'ordonnançabilité
– l'obtention d'une séquence d'ordonnancement

Comme indiqué sur le schéma synoptique suivant (figure 2.2 tiré de [Kop97]), si un test d'ordonnançabilité s'appuyant sur une condition suffisante d'ordonnançabilité produit un résultat positif, le système de tâches est définitivement ordonnançable. Par contre, si le résultat est négatif, le système peut être ordonnançable ou non. De la même manière, si un test d'ordonnançabilité s'appuyant sur une condition nécessaire d'ordonnançabilité produit un résultat positif, l'ensemble de tâches peut ne pas être ordonnançable alors que s'il est négatif, le système de tâches est définitivement non ordonnançable.

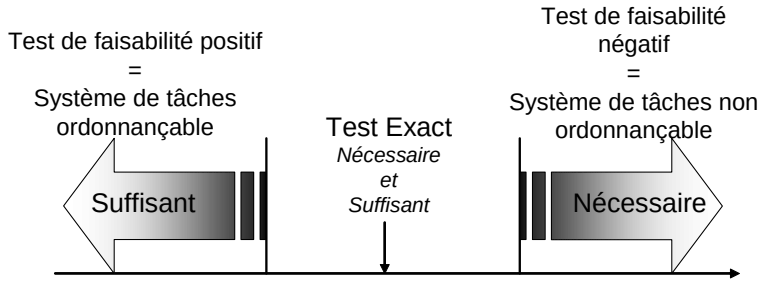

Fig. 2.2: Synoptique des différents résultats des tests d'ordonnançabilité.

Nous nous intéressons ici, à la complexité [Coo71, Coo83, GJ79] de quelques problèmes pertinent pour l'ordonnancement formant la majeure partie des différentes classe des problèmes. La difficulté d'obtention d'une condition d'ordonnançabilité dépend de la classe du problème étudié. Le tableau suivant énumère les différentes classes de problèmes les plus courants en théorie de l'ordonnancement Temps Réel ainsi que la complexité associée au problème de l'ordonnançabilité.

PRÉEMPTIF				
tâches à départ :	*instructions conditionnelles*	*échéances sur requête*	*existence d'un ordonnancement*	*référence*
simultanés	non	oui	polynomiale	ED cf.2.3.2.1
simultanés	non	non	problème ouvert algo. pseudo-polynomiale	[BHR90]
différés	non	indifférent	co-NP-difficile	[BHR90]
indifférents	oui	indifférent	NP-difficile	[CET01]
Le cas avec *partage de ressources* est un cas particulier du *non préemptif*				
NON PRÉEMPTIF				
départs simultanés	échéances inférieures à la période		NP-difficile	[Mok83]
départs différés			NP-difficile	[GJ79, JSM91]

2.3 Résultats de l'approche En-Ligne

La majorité des ordonnanceurs en ligne repose sur la notion de priorité. L'attribution des priorités des tâches peut être effectuée hors ligne, c'est à dire que les prioritées sont calculées avant la mise en service du système. Dans ce cas, chaque tâche garde une même priorité tout au long du cycle de vie du système. L'affectation de priorités peut également être effectuée en ligne. Dans ce cas, l'ordonnanceur exécute un algorithme spécifique chargé d'évaluer une priorité pour chaque tâche active à chaque fois qu'il est sollicité. Ainsi, une même tâche peut obtenir différentes priorités durant son exécution. Cette allocation de priorités plus complexe nécessite des services ou routines supplémentaires (comme la gestion des files de tâches actives par exemple) ce qui augmente le temps d'exécution effectif de l'ordonnanceur. Dans le premier cas, nous parlons d'algorithme d'*ordonnancement à priorités fixes* contrairement à la deuxième méthode appelée *ordonnancement à priorités dynamiques*.

Nous nous intéressons dans un premier temps aux systèmes de tâches ne comportant pas de contraintes de précédence et ne partageant pas de ressources.

2.3.1 Algorithmes d'ordonnancement à priorité fixes

De façon générale, les analyses d'ordonnançabilité dans le cas des algorithmes à priorités fixes, reposent sur des critères analytiques. Or, il est inutile de rechercher une faute temporelle lorsque le processeur est inutilisé. Ceci nous amène à définir la notion de période d'activité du processeur. Nous pouvons noter que les intervalles de temps où le processeur est occupé ou libre sont les mêmes pour tous les algorithmes d'ordonnancement conservatifs. Par ailleurs la plus longue période d'activité du processeur survient lorsque toutes les tâches sont activées simultanément. Un tel évènement est appelé un *instant critique* [LL73] et correspond intuitivement à un instant où les tâches sont dans la pire configuration possible par rapport au problème de l'ordonnancement. Ainsi, une tâche aura le pire temps de réponse si toutes les tâches plus prioritaires qu'elle sont activées en même temps qu'elle.

Définition 9 ***[LL73] Instant critique***

Un ***instant critique*** *est un instant t auquel toutes les tâches sont activées simultanément.*

Par conséquent, les critères analytiques pour l'analyse d'ordonnancement des algorithmes à priorité fixes reposent sur cette notion d'instant critique. Dans le cas de tâches à départs simultanés, l'instant critique se produit au temps 0 puisque le processeur doit exécuter l'ensemble des tâches à cette date.

Nous présentons par la suite les principaux algorithmes à priorités fixes dans un contexte monoprocesseur. Un état de l'art est présenté dans [ABD+95].

2.3.1.1 Rate Monotonic

Cet algorithme noté RM, a été défini par [Ser72, LL73]. Il est basé sur une affectation de priorités inversement proportionnelles à la période des tâches. En d'autres termes, la priorité la plus forte est attribuée à la tâche possédant la période la plus petite. Aucune règle n'est prévue lorsque plusieurs tâches ont une même période. Les résultats suivant ont été établis par [LL73]. La figure 2.3 illustre le fonctionnement de cet algorithme.

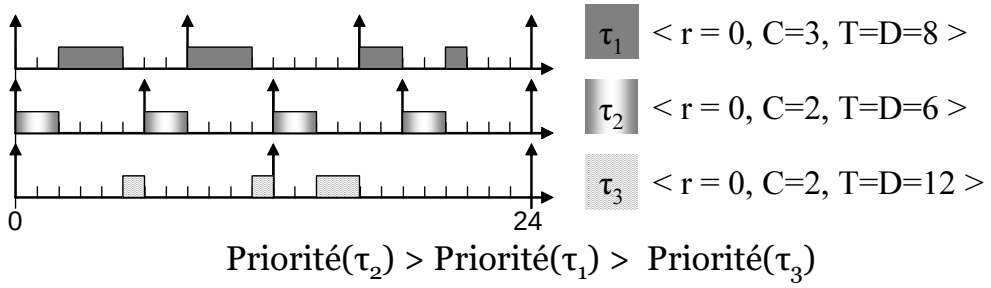

Fig. 2.3: Séquence d'ordonnancement suivant RM. On note que la charge processeur est de $\frac{3}{8}+\frac{2}{6}+\frac{3}{12}=0,875$ et que donc la condition suffisante de [LL73] n'est pas respectée : $3(2^{1/3}-1)\approx 0,77976<0,875$.

Théorème 1 ***[LL73] Optimalité de Rate Monotonic***

L'algorithme RM est optimal dans la classe des algorithmes à priorités fixes et dans un contexte de tâches périodiques indépendantes à départs simultanés et à échéances sur requête.

Nous disposons de critères analytiques l'ordonnançabilité d'un système de tâche par l'algorithme RM reposant sur la notion d'instant critique. En effet, [LL73] ont établi une Condition Suffisante(CS) basée sur la pire configuration possible pour l'ordonnancement des tâches. Dans le cadre de tâches à départs simultanées, l'instant critique s'obtient au temps 0 et la période d'activité du processeur qui s'en suit correspond à une charge maximale du processeur. Ainsi, à l'instant correspondant au $Max(T_i)$ les n tâches doivent s'être exécutées au moins une fois et ce sans aucune faute temporelle. Nous en tirons la CS suivante :

Théorème 2 ***[LL73] Condition Suffisante d'ordonnançabilité de RM***

Soit une application temps réel composée de n tâches périodiques indépendantes et à échéances sur requête. Si

$$U=\sum_{i=1}^{n}\frac{C_i}{T_i}\leq n(2^{1/n}-1)$$

alors l'application est ordonnançable par Rate Monotonic.

Ce critère analytique d'ordonnançabilité tend (en décroissant) vers $ln\ 2\ (0,69)$ lorque n tend vers l'infini. Or, ce seuil est relativement strict comparé aux études expérimentales (basées sur des configurations de tâches aléatoires) donnant un seuil à la charge processeur acceptable par RM de l'ordre de 85%.

Une Condition Nécessaire et Suffisante(CNS) d'ordonnançabilité a également été établie par [LSD89] dans le même contexte. Mais, cette CNS souffre, malgré une étude plus détaillée de l'instant critique, d'une complexité pseudo-polynomiale ce qui la rend peu performante. En effet, son calcul revient à peu de chose près à une effectuer une simulation.

Théorème 3 ***[LSD89] Condition Nécessaire et Suffisante d'ordonnançabilité de RM***

Soit une application temps réel composée de n tâches périodiques indépendantes à départs simultanés et à échéances sur requête avec $T_1 < \cdots < T_n$ alors l'application est ordonnançable par RM si, et seulement si :

$$\forall i,\ 1 \leq i \leq n \qquad min_{t \in S_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \leq 1$$
$$avec \qquad S_i = \left\{ kT_j, 1 \leq j \leq i, k = 1, \cdots, \left\lceil \frac{T_i}{T_j} \right\rceil \right\}$$

2.3.1.2 Deadline Monotonic

Nous venons de voir RM qui est optimal dans son contexte. Or, ce dernier ne s'étend pas aux tâches à échéances inférieures aux périodes. Leung et Whitehead en 1982 [LW82] ont donc présenté un algorithme nommé *Deadline Monotonic* ou *Inverse Deadline* basé sur une affectation de priorités inversement proportionnelles aux délais critiques des tâches. Ainsi, une tâche τ_i de délai critique D_i sera plus prioritaire qu'une tâche τ_j de délai critique D_j si $D_i < D_j$. En cas d'égalité, rien n'est spécifié comme dans le cas RM, ces conflits sont donc résolus de façon statique (un ordre de priorité est défini préalablement). Lorsque les tâches sont à échéances sur requête DM équivaut à RM. La figure 2.4 illustre une séquence d'ordonnancement suivant DM.

Comme RM, DM est optimal dans son contexte.

Théorème 4 ***[LW82] Optimalité de Deadline Monotonic***

L'algorithme DM est optimal dans la classe des algorithmes à priorités fixes et dans un contexte de tâches périodiques indépendantes à départs simultanés et à échéances inférieures aux périodes.

Nous pouvons noter que [Leh90] a montré la non-optimalité de DM dans le contexte de tâches à échéances non reliées à la période (et plus précisément dans le cas $D_i > T_i$). Une condition suffisante d'ordonnançabilité a été mis en évidence par [LW82].

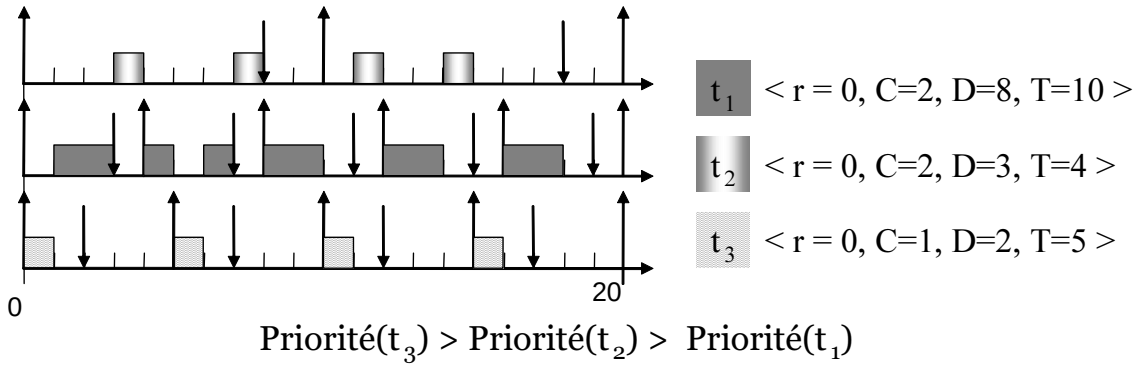

FIG. 2.4: Séquence d'ordonnancement suivant DM. Nous pouvons noter que le critère analytique de [LW82] est très pessimiste. $\sum_{i=1}^{n} \frac{C_i}{D_i} = \frac{2}{8} + \frac{2}{3} + \frac{1}{2} \approx 1,4167$ et la condition suffisante de [LW82] n'est pas respectée : $3(2^{1/3} - 1) \approx 0,77976 < 1,4167$, et pourtant cette configuration de tâche est ordonnançable par DM.

THÉORÈME 5 ***[LW82] Condition Suffisante d'ordonnançabilité de DM***

Soit une application temps réel composée de n tâches périodiques indépendantes et à échéances inférieures aux périodes. Si

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1)$$

alors l'application est ordonnançable par Deadline Monotonic.

Le test de [LSD89] a été étendu pour permettre l'obtention d'une condition nécessaire et suffisante d'ordonnançabilité pour DM.

THÉORÈME 6 ***[LSST91] Condition Nécessaire et Suffisante d'ordonnançabilité de DM***

Soit une application temps réel composée de n tâches périodiques indépendantes à départs simultanés et à échéances inférieures aux périodes avec $D_1 < \cdots < D_n$ alors l'application est ordonnançable par DM si, et seulement si :

$$\forall i,\ 1 \leq i \leq n \qquad min_{t \in S_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \leq 1$$

$$\text{avec} \qquad S_i = \{D_i\} \cup \left\{ kT_j, 1 \leq j \leq i, k = 1, \cdots, \left\lceil \frac{D_i}{T_j} \right\rceil \right\}$$

Comme pour RM, la complexité de ce test est pseudo polynomiale. Toutefois, nous pouvons trouver de nombreuses études sur DM qui améliorent ces résultats : [ABRW92, MA98, SS93, Har87, Jos85, JP86, ABRW91].

2.3.1.3 *Affectation des priorités suivant l'algorithme d'Audsley*

Nous avons vu que Rate Monotonic et Deadline Monotonic étaient optimaux dans leur contexte respectif d'ordonnançabilité. Dans le cas de tâches indépendantes à départs différés ou/et à échéances non reliées aux périodes, ils ne le sont plus [Leh90]. Dans ce contexte plus général, Audsley [Aud91] propose un algorithme optimal basé sur une affectation incrémentale de priorités de la moins prioritaire à la plus prioritaire.

Cet algorithme tire s'appuie sur les propriétés des temps de réponse. En effet, Audsley a constaté qu'une diminution du niveau de priorité d'une tâche n'engendre pas d'augmentation du temps de réponse des autres tâches. De plus, une tâche de priorité P_k est insensible à l'ordre des affectations des tâches plus prioritaires qu'elle. L'algorithme d'Audsley tire ainsi profit de ces constatations et affecte les priorités des tâches de manière incrémentale de la plus faible à la plus forte. L'algorithme détermine en premier lieu la tâche de plus faible priorité en effectuant un test d'ordonnançabilité co-NP-difficile basé sur le temps de réponse (en utilisant l'instant critique sachant que les autres tâches sont plus prioritaires) permettant de vérifier que cette affectation conduira à un ordonnancement valide. Une fois que cette priorité est affectée à une tâche, il faut recommencer la démarche avec la plus faible priorité des priorités non allouées aux tâches et ainsi de suite. Une fois que toutes les tâches ont reçu une priorité, la séquence d'ordonnancement produite sera nécessairement valide. Le résultat de cet algorithme est alors une condition nécessaire et suffisante d'ordonnançabilité dans la classe des algorithmes à priorités fixes.

Théorème 7 ***[Aud91] Optimalité de l'algorithme d'Audsley***

L'algorithme d'affectation des priorités d'Audsley est optimal dans la classe des algorithmes à priorités fixes pour une application temps réel composée de tâches indépendantes à échéances non reliées aux périodes et à départs différés.

Cette analyse d'ordonnançabilité souffre malgré son caractère optimal d'une complexité importante de par sa construction incrémentale et du test d'ordonnançabilité co-NP-difficile.

2.3.2 *Algorithmes d'ordonnancement à priorités dynamiques*

Nous venons de voir que les algorithmes d'ordonnancement à priorité fixes RM et DM étaient optimaux chacun dans leur classe. Dans le cas de tâches à départs différés seul l'algorithme d'Audsley peut être appliqué, or, comme nous l'avons vu celui ci souffre d'une complexité de mise en oeuvre importante. Pour palier ces problèmes, les algorithmes d'ordonnancement à priorités dynamiques offrent de nouvelles opportunités en terme de puissance d'ordonnançabilité. À la différence des algorithmes à priorité fixes, ici, l'affectation des priorités n'est pas définitive. Une même tâche (et parfois une même instance de tâche) peut recevoir des priorités différentes durant la vie de l'application. La priorité est ici une fonction du temps et elle dépend de l'état (instantané) du système à chaque instant. De façon claire, ce type de méthodes présente des limites, liées à la complexité

éventuelle de leur mise en oeuvre : en effet, les calculs des priorités induisent un surcoût (ou overhead) de temps processeur à chaque exécution de l'ordonnanceur. S'ils sont trop complexes, les algorithmes les mettant en oeuvre seront pénalisants pour le système. Nous présentons ici les deux principaux algorithmes d'ordonnancement à priorités dynamiques : Earliest Deadline et Least Laxity dont la complexité de calcul des priorités est faible (linéaire en le nombre de tâches).

2.3.2.1 Earliest Deadline

Le principe de fonctionnement de cet algorithme repose sur les travaux de [Jac55] sur les *job shop*. La mise en évidence du principe de l'exécution des tâches les plus urgentes, a permis à [Ser72] et [LL73] d'adapter ce résultat au contexte temps réel en introduisant les algorithmes initiaux respectivement *Relative Urgency* et *Deadline Driven Scheduling* qui prendront le nom ultérieurement de *Earliest Deadline*. Cet algorithme est basé sur les échéances des tâches actives à chaque instant. Ainsi, la tâche la plus prioritaire est celle donc l'échéance est la plus proche. En cas d'égalité, c'est la plus ancienne dans la file d'attente qui est choisie. La figure 2.5 illustre le fonctionnement de Earliest Deadline.

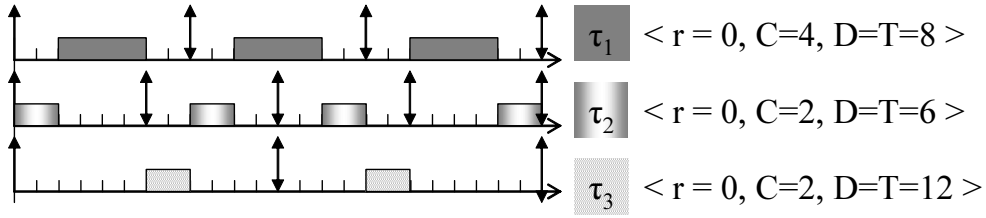

Fig. 2.5: Séquence d'ordonnancement suivant ED. On note que $\sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{4}{8} + \frac{2}{6} + \frac{2}{12} = 1$.

Cet algorithme possède une puissance d'ordonnancement intéressante lorsque l'on considère des tâches indépendantes :

Théorème 8 ***[Der74, Lab74] Optimalité de ED***

L'algorithme Earliest Deadline est optimal pour une application temps réel composée de tâches indépendantes à échéances inférieures ou égales aux périodes.

[LL73] ont mis en évidence une condition nécessaire et suffisante d'ordonnançabilité pour ED. Cette dernière est la plus permissive qu'il soit puisqu'elle s'appuie sur une utilisation maximale du processeur (U=100%).

Théorème 9 ***[LL73] Condition nécessaire et suffisante d'ordonnançabilité de ED***

Une application temps réel composée de n tâches indépendantes à départs simultanés et à échéances sur requête est ordonnançable par Earliest Deadline si et seulement si :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

.

Dans le contexte de tâches indépendantes à départs simultanés tels que $D_i \geq T_i$, [BMR90] ont étendu le résultat du théorème 9. Dans le cas général de tâches indépendantes, seule une simulation sur une durée d'étude prédéterminée permet de conclure sur la faisabilité de ED. Cette durée d'étude a été étudiée et bornée par [LM80, LW82] puis par [BHR90, RCM96, ZS94] puis minimisé par [CGG04]. Nous regarderons plus précisément les problèmes de durée de simulation en 2.4. Nous pouvons noter également que ED a été comparé aux autres algorithmes comme RM et DM en ce qui concerne le sur-coût occasionné :[Hen75] a montré que ED minimise le nombre de changements de contexte. Par ailleurs, les algorithmes en eux-même sont tous de complexité similaire (linéaire en le nombre de tâches).

2.3.2.2 Least Laxity

L'algorithme Least Laxity est basé sur une affectation de priorités en fonction de la laxité dynamique (cf. Définition 3). Ainsi, la tâche à exécuter à un instant t est celle dont la laxité dynamique est la plus petite c'est à dire celle dont l'exécution peut le moins être retardée. Comme pour RM et DM, rien n'a été spécifié dans le cas de tâche ayant la même laxité. De par la définition même de LL, le calcul des priorités doit être effectué à chaque modification de laxité, donc à chaque exécution d'une unité de temps ce qui engendre un surcoût processeur conséquent. De plus, comme l'illustre la figure 2.6, LL génère un nombre important de changements de contexte.

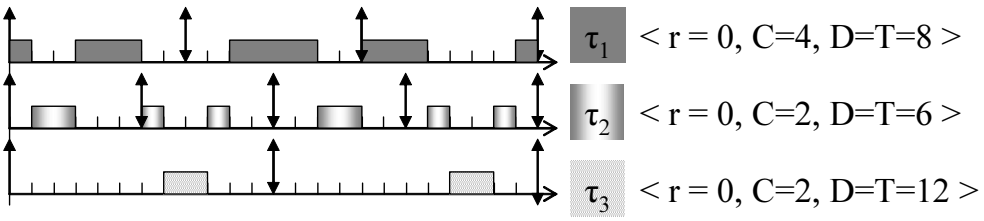

Fig. 2.6: Séquence d'ordonnancement suivant LL. On note que $\sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{4}{8} + \frac{2}{6} + \frac{2}{12} = 1$. Nous pouvons remarquer une augmentation du nombre de changement de contexte en comparaison avec la figure 2.5 représentant la même application ordonnancée suivant ED (8 changements de contexte pour ED et 12 pour LL).

Cet algorithme possède la même puissance d'ordonnançabilité que ED :

Théorème 10 ***[DM89, Mok83] Optimalité de LL***

L'algorithme Least Laxity est optimal pour une application temps réel composée de tâches indépendantes à échéances inférieures ou égales aux périodes.

LL n'est toutefois pas très intéressant en monoprocesseur comparé à ED qui engendre moins de changements de contexte. Par contre, LL devient plus performant que ED en multiprocesseur où il profite d'une plus grande puissance d'ordonnançabilité.

2.3.3 Tâches dépendantes

Nous venons de voir des algorithmes d'ordonnancement basés sur l'affectation de priorités pour des tâches indépendantes. La plupart des systèmes temps réel nécessitent cependant fréquemment le partage de ressources ou la transmission de données entre les tâches. Nous présentons maintenant les modifications à apporter aux algorithmes précédents et les protocoles à leur adjoindre pour qu'ils puissent fonctionner dans un contexte de tâches dépendantes (ou couplées).

2.3.3.1 Contraintes de précédences

Nous parlons de contraintes de précédence pour désigner toute dépendance entre les tâches. Il s'agit donc de définir une dépendance temporelle entre les tâches exprimant :

- soit le fait qu'une tâche doit avoir terminée tout (ou partie) de son activité avant qu'une autre tâche puisse poursuivre son exécution.
- soit qu'une tâche attende un message émis par une autre tâche.

Une première Condition Nécessaire d'ordonnançabilité peut être aisément mise en évidence. En effet, une tâche réceptrice ou en attente de synchronisation doit recevoir l'information avant son échéance par conséquent, la tâche émettrice doit envoyer l'information à la même fréquence sous peine de voir la tâche réceptrice dépasser son échéance ou risquer un écrasement d'informations ou de saturation du buffer et dans ce cas , c'est la tâche émettrice qui ratera son échéance. Ainsi nous considérons que les tâches liées par une contrainte de précédence doivent avoir une période identique.

Blazewicz [Bla76] a proposé une méthode permettant de rendre ED optimal en présence de précédences. Pour cela, il est nécessaire de transformer les tâches communiquantes en sous-tâches de façon à ne garder qu'une unique émission ou réception par tâche respectivement en fin ou début de tâche. L'ensemble de ces sous-tâches est muni d'un ordre partiel $\prec$ tel que si $\tau_i \prec \tau_j$ alors la tâche τ_i doit s'exécuter avant τ_j. Ce découpage appelé *mise en forme normale* a permis à [Bla76] de proposer une modification des paramètres r_i et D_i afin d'ordonner l'exécution des sous-tâches en accord avec les contraintes de précédence. La prise en compte de ces modifications s'opère en $O(n^2)$.

[SS94] a étendu ce résultat pour ED en rajoutant la contrainte supplémentaire du partage de ressource, tant que les dates r_i et D_i respectent l'ordre partiel établi pour respecter les contraintes. Il existe également une méthode [Tin76] pour les algorithmes à

priorités fixes basée sur un décalage des tâches réceptrices d'un temps équivalent au pire temps de réponse de la tâche émettrice.

[RC99, RCR01] ont proposé deux approches pour traiter ce type de contraintes dans un contexte différent où les périodes des tâches émettrices et réceptrices ne sont pas identiques : c'est ce que nous appelons les *contraintes de précédence généralisée*. Ces méthodes sont basées soit sur une modification des paramètres des tâches de façon à les rendre indépendantes des contraintes de précédence (non applicable avec RM), soit en dupliquant certaines tâches de façon à se ramener à un graphe de tâches provenant de l'ordre partiel inhérent aux précédence, soumises à des contraintes de précédence simples. Dans ce dernier cas, les résultats de contraintes de précédence simples sont ensuite appliqués mais le nombre de tâches n'est plus borné polynomialement puisqu'il y a duplication de tâche.

2.3.3.2 Partage de ressources

Au même titre que les contraintes de précédence, la conception de système temps réel n'utilisant pas de ressources reste extrêmement rare. Or, l'utilisation de ressources engendre des difficultés liées à la protection des accès aux ressources par des sections critiques. En effet, en temps réel, il est impératif de maîtriser le temps nécessaire à chaque tâche pour terminer son exécution. Cependant la mise en concurrence de plusieurs tâches pour l'accès à une ressource engendre des retards d'exécution. [Mok83] a montré que la problème d'ordonnancement en présence de ressources est NP-difficile au sens fort, et qu'il n'existe pas d'algorithme optimal dans le contexte général. De plus, le partage de ressources engendre deux phénomènes critiques :

- *l'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. Pour exemple, prenons le cas où une tâche τ_i détient la ressource R1 et demande la ressource R2 pour continuer son exécution. Si R2 est détenue par une autre tâche τ_j et que celle-ci demande la ressource R1 pour se terminer, alors les deux tâches ne peuvent plus être exécutées et le système est définitivement bloqué.
- *l'inversion de priorité* se produit lorsqu'une tâche τ_1 plus prioritaire demande une ressource détenue par une tâche τ_2 moins prioritaire. Cette dernière est logiquement exécutée pour libérer la ressource. Mais il peut arriver qu'une troisième tâche τ_3 plus prioritaire que τ_2 mais moins que τ_1, préempte naturellement τ_2 sans utiliser de ressource. Le système est donc amené à exécuter τ_3 avant τ_1 ce qui va à l'encontre des règles de priorité. Cette inversion de priorité peut être non bornée dans le temps si toutes les tâches de priorités comprises entre celles de τ_2 et τ_1 sont exécutées. L'exemple de la mission MARS Pathfinder [CDKM00, Cot99] a illustré aux dépends de la NASA, le dysfonctionnement occasionné par cet type de phénomène.

Pour palier ces problèmes, des protocoles de gestion des ressources ont été mis au point pour fonctionner avec les algorithmes vus précédemment. Les deux premiers protocoles sont issus de la notion de *superpriorité* proposée par [Kai82].

· Protocole à Priorité Héritée (PPH)

Ce protocole (appelé aussi Priority Inheritance Protocol) [SRL87, SRL90] a été défini préalablement pour les algorithmes d'ordonnancement à priorités fixes. Son fonctionnement est basé sur l'évitement des inversions de priorités. Quand une tâche τ_i de priorité $Prio_i$ bloque l'exécution d'une tâche τ_j de priorité $Prio_j > Prio_i$ (ie. supérieure), alors τ_i hérite de la priorité $Prio_j$ de τ_j jusqu'à ce qu'elle relâche les ressources pour lesquelles τ_j est bloquée. Nous notons que cet héritage de priorité est transitif, ce qui se traduit par une chaîne d'héritages de priorités. L'héritage de priorité induit deux types de blocage, appelé *blocage direct* (Direct Blocking) et *blocage transitif* (Push-through blocking). Dans le premier cas, une tâche τ_i peut être bloquée par une tâche τ_j de priorité inférieure si τ_j a commencé sa section critique avant le réveil de τ_i. Le blocage transitif survient lorsque la ressource utilisée par τ_j lui a fait hériter d'une priorité plus élevée que celle de τ_i. Ces deux blocages permettent ainsi d'éviter le phénomène d'inversion de priorités. Par contre, ce protocole ne permet pas d'éviter l'interblocage. Le temps de blocage B_i d'une tâche τ_i par PPH a été évalué par [SRL90, SRL87, But97, Liu00] et reste extrêmement pessimiste :

$$B_i = (\text{nombre d'instances des tâches } \tau_j \text{ pouvant bloquer } \tau_i \text{ tel que } prio(\tau_i) < prio(\tau_j)\ ) \times (\text{durée maximale des sections critiques})$$

· **Protocole à Priorité Plafond (PPP)**

Afin de palier les inconvénients de PPH, [SRL87, SRL90] ont également mis au point le Protocole à Priorité Plafond (appelé Priority Ceiling Protocol en anglais) qui permet d'éviter non seulement l'inversion de priorité mais également l'interblocage pour les algorithmes à priorité fixe ([CL90] a étendu à ED ce protocole (*protocole à priorité plafond dynamique PPPD*). Le principe est d'associer à chaque ressource un seuil de priorité égal à la plus grande priorité parmi toutes les tâches qui peuvent la demander. Ainsi, lorsqu'une tâche demande une ressource, elle n'est autorisée à la prendre que si sa priorité est supérieure aux seuils de priorité de toutes les ressources détenues par d'autres tâches. Si la tâche demandeuse est bloquée, elle transmet sa priorité par héritage à la tâche qui possède la ressource demandée. La transitivité de l'héritage de priorités est mise en œuvre comme pour PPH.

[SRL90] ont montré que le temps de blocage induit par ce protocole est au plus égal à la durée de la plus longue section critique, ce qui est meilleur que ce que l'on avait pour le protocole PPH. Ce temps de blocage maximum B_i de la tâche τ_i induit par le partage de ressource correspond donc à la plus longue section critique parmi les tâches moins prioritaires que τ_i et dont la priorité plafond est supérieure à la priorité de τ_i. En posant, $SC_{j,k}$ la durée maximale de la section critique de la tâche τ_j pour la ressource R_k, PP_{R_k} la priorité plafond de la ressource R_k, alors le pire temps de blocage d'une tâche τ_i s'exprime par

$$B_i = max_{j,k}\{SC_{j,k} \textit{ tel que } Prio(\tau_j) < Prio(\tau_i) \textit{ et } PP_{R_k} \geq Prio(\tau_i)\}$$

· **Protocole d'Allocation de la Pile (PAP)**

Ce protocole proposé par [Bak91] permet de s'abstraire du type d'allocation de

priorité des tâches (à savoir dynamique ou fixe). Ainsi, ED est naturellement pris en compte sans contraintes particulières. Le principe de ce protocole repose sur une diminution du nombre de préemptions dues au blocage pour le partage de ressource et par conséquent le nombre de changements de contexte. Pour ce faire, une tâche n'est pas autorisée à démarrer tant que toutes les ressources qui lui sont nécessaires ne sont pas disponibles. Ce qui évite les interblocages. Tout blocage est ainsi banni une fois qu'elle a démarré son exécution. La mise en œuvre de ce protocole repose sur les notions suivantes :

- un niveau de préemption noté π_i, fixé hors ligne et indépendant du type d'ordonnancement (à priorité fixe ou dynamique) permet de prédire les éventuels blocages. Ce niveau de préemption peut être ordonné par la priorité classique des tâches suivant leur date d'arrivée (en général, nous utilisons les priorités suivant DM). Ainsi, si τ_i arrive avant τ_j et que $Prio(\tau_i) > Prio(\tau_j)$ alors $\pi_j > \pi_i$.
- la valeur plafond d'une ressource C_{R_k}, qui est dynamique et est déterminée par la valeur maximale des priorités des tâches actives ou susceptibles de préempter la tâche active nécessitant une instance de la ressource R_k.
- le seuil de priorité Π courant, dynamique également, correspond au maximum des $C_{R_k}, \forall k$. (ce protocole est déjà utilisé dans le protocole PPP).

Le fonctionnement de PAP est relativement simple à mettre en œuvre à partir des paramètres précédent. Une tâche τ_i peut préempter une autre tâche τ_j lorsqu'elle souhaite prendre une ressource non disponible si :

- elle est la plus prioritaire parmi les tâches actives,
- $\pi_i \geq \Pi$.

Ce protocole permet de s'assurer que l'exécution d'une tâche ne peut se faire que si toutes les ressources nécessaires sont libres (c'est également le cas pour les tâches susceptibles de la bloquer). De plus, le principe d'héritage est gardé comme pour PHP par le biais du paramètre Π et le fonctionnement, en considérant les instances de ressources libres, permet l'utilisation de ressources multi-instances. [Bak91] a montré que PAP gardait les mêmes propriétés que PPP à savoir, l'évitement des interblocages et des inversions de priorité et que le nombre de blocages de chaque tâche était limité à un et de durée au plus la durée d'une section critique.

Ces différents protocoles permettent de dégager une borne du temps de blocage B_i pour chaque tâche en accord avec l'algorithme d'ordonnancement à priorité choisi. Valider l'application revient alors à intégrer dans les conditions de faisabilité ce paramètre B_i. Ainsi, pour l'algorithme ED [SS93] et [Bak91] nous avons la CS suivante :

$$\forall i \in [1..n], \left(\sum_{k=1}^{i} \frac{C_k}{min(T_k, D_k)} \right) + \frac{B_i}{min(T_i, D_i)} \leq 1$$

et pour les algorithmes à priorités fixes ([SRL90] pour RM) une CS :

$$\forall i \in [1..n], \left(\sum_{k=1}^{i} \frac{C_k}{min(T_k, D_k)} \right) + \frac{B_i}{min(T_i, D_i)} \leq i \left(2^{1/i} - 1\right)$$

Nous notons qu'il existe également des conditions reposant sur une analyse du temps de réponse au pire cas et de la demande processeur. Pour de plus amples détails sur ces techniques, nous pouvons nous référer à [Ric03, Liu00, But97].

2.3.4 Anomalies d'ordonnancement

La prise en compte des contraintes précédence et de partages de ressources en ordonnancement monoprocesseur à priorité fixe peut engendrer des anomalies d'ordonnancement. En effet, [Gra69] a mis en évidence en environnement multiprocesseur des anomalies d'ordonnancement illustrées dans [SSDB94, But97]. De cette constatation, [Ric02, Gro99] ont montré que ces anomalies étaient également présentes avec des tâches périodiques à priorités fixes. L'exemple de la figure 2.7 illustre de tel phénomène. On constate qu'une anomalie d'ordonnancement sous Deadline Monotonic survient en raison d'une réduction de la durée d'exécution de la tâche τ_3. En effet, τ_1 et τ_2 partagent la même ressource, par conséquent elles ne peuvent pas se préempter mutuellement durant l'utilisation de la ressource. Nous constatons alors que dans le deuxième cas, l'exécution de τ_1 avant τ_2 au temps 7 (en raison de la réduction de la durée de τ_3) entraîne un dépassement de l'échéance de τ_2.

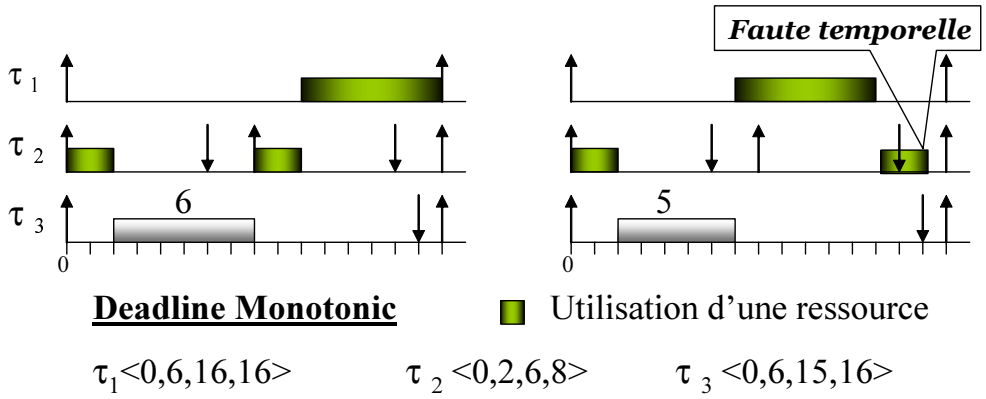

Fig. 2.7: Exemple d'une anomalie d'ordonnancement en raison d'une réduction de la durée d'exécution de la tâche τ_3

Propriété 1 ***[Gen00]Anomalies d'ordonnancement***

Un système de tâche ordonnançable avec les pires durées d'exécution peut être non ordonnançable avec des durées réelles d'exécution plus petites.

En environnement monoprocesseur, les systèmes de tâches non préemptifs (qui est pour rappel, un cas particuliers d'un système avec partage de ressources en exclusion mutuelle) à priorités fixes avec contraintes de précédence ou encore avec suspension (durant une opération d'entrée/sortie par exemple) sont soumis à ces anomalies d'ordonnancement.

Il existe également des méthodes d'attribution de priorités comme dans [Ric02] permettant de s'affranchir de ce type d'anomalies. Nous parlons alors d'*ordonnanceur robuste* si l'ordonnancement n'est pas assujetti à des anomalies d'ordonnancement pour un système de tâches donné.

2.3.5 Tâches apériodiques et sporadiques

Les études effectuées sur l'ordonnancement des tâches non périodiques sont basées sur des hypothèses contraignantes pour l'ordonnançabilité de l'application temps réel, mais celles-ci sont nécessaires pour valider les résultats obtenus. Ces suppositions sont les mêmes que celles utilisées par [LL73], pour leur description des algorithmes d'ordonnancement RM et ED. Toutes les méthodes que nous présenterons ultérieurement sont basées sur l'étude des applications temps réel composées de tâches indépendantes à départs simultanés et à échéances sur requête. Les tâches apériodiques et sporadiques à ordonnancer peuvent être critiques ou non.

La prise en compte par le processeur de ce type de tâches s'opère suivant deux approches :

- par un *test d'acceptation* : chaque arrivée d'une nouvelle instance de tâches sporadiques/apériodiques nécessite d'être évaluée par un algorithme particulier qui déterminera si cette instance peut ou non être exécutée par le processeur selon que sa prise en compte engendre ou non un dépassement d'échéance des tâches périodiques.
- par un *ordonnancement conjoint* : chaque instance de tâches sporadiques/apériodiques est traitée conjointement aux tâches périodiques mais en usant de régles d'ordonnancement différentes pour ne pas provoquer de dépassement des échéances des tâches périodiques.

Quelque soit l'approche utilisée, les objectifs d'ordonnancement pour ces tâches peuvent être de deux ordres : soit garantir leur échéance, soit minimiser leur temps de réponse. Généralement, nous utilisons les tests d'acceptation pour les tâches critiques et préférons réduire les temps de réponses pour les tâches souples avec un ordonnancement conjoint.

Les études menées par [TLS96] définit les critères d'optimalité pour la gestion des tâches non périodiques.

Définition 10 ***[TLS96] Optimalité de l'ordonnancement de tâches non périodiques***

- *Un algorithme est* ***fortement optimal*** *s'il garantit la configuration des tâches périodiques et s'il minimise le temps de réponse de chaque tâche apériodique.*
- *Un algorithme d'ordonnancement est* ***optimal*** *si c'est un algorithme qui garantit la configuration des tâches périodiques et qui minimise le temps de réponse moyen des tâches apériodiques.*

Si un algorithme est fortement optimal alors il n'existe pas d'autres algorithmes qui respectent les contraintes des tâches périodiques et qui permettent d'obtenir un temps de

réponse inférieur d'au moins une des tâches apériodiques. Les algorithmes qui décrivent des situations idéales, n'existent malheureusement pas, comme l'illustre la propriété suivante :

Propriété 2 ***[TLS96] Non existence d'ordonnancement optimal de tâches non périodiques***

Pour toute configuration de tâches périodiques ordonnancées par un algorithme à priorités fixes et un flot quelconque de tâches apériodiques
– *il n'existe aucun algorithme d'ordonnancement fortement optimal,*
– *il n'existe aucun algorithme dynamique optimal.*

Malgré ces résultats, certains algorithmes permettent une gestion des tâches non périodiques. Ces derniers peuvent être classés suivant deux catégories :

– La première utilise une tâche serveur dédiée uniquement au service apériodique : les méthodes sont dites de *type serveur*. La technique du serveur consiste en la création d'une tâche périodique supplémentaire. Elle possède comme les autres, une période T_s et une durée d'exécution appelée C_s. Son fonctionnement consiste à distribuer sa capacité C_s aux tâches apériodiques actives, c'est à dire ayant été activées et en attente d'exécution. Cette méthode ne remet pas en cause l'ordonnançabilité des tâches périodiques, mais le taux d'utilisation du processeur doit tenir compte de cette tâche supplémentaire pour l'analyse d'ordonnançabilité. Ainsi en prenant un ensemble de n tâches périodiques et une tâche serveur τ_s, le taux d'utilisation total du processeur U_t est la somme du taux d'utilisation des n tâches plus la charge processeur de la tâche serveur soit :

$$U_t = U + U_s = \sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C_s}{T_s}$$

  Lorsque plusieurs tâches apériodiques sont en attente de la capacité de la tâche serveur, la politique FIFO [1] est généralement utilisée mais on peut envisager d'autres techniques implémentant des notions de propriétés par exemple.
– La deuxième consiste à récupérer les temps creux de l'ordonnancement périodique pour servir au mieux les tâches apériodiques : ces méthodes sont dites de type *vol de temps creux*. Elles sont basées sur une politique de gestion des temps d'inactivité processeur résultant de la charge du système de tâches périodiques. Les tâches non périodiques sont en général stockées dans une seconde file de tâches comme l'illustre la figure 2.8, une fois activées dans le système, et exécutées dès qu'il apparaît un temps d'inactivité processeur.

Pour chacune de ces méthodes, il est courant de distinguer :
– L'algorithme d'ordonnancement de la configuration des tâches périodiques à savoir à priorités fixes ou dynamiques. Nous ne considérons pour cette étude que les deux

[1] Fisrt In First Out

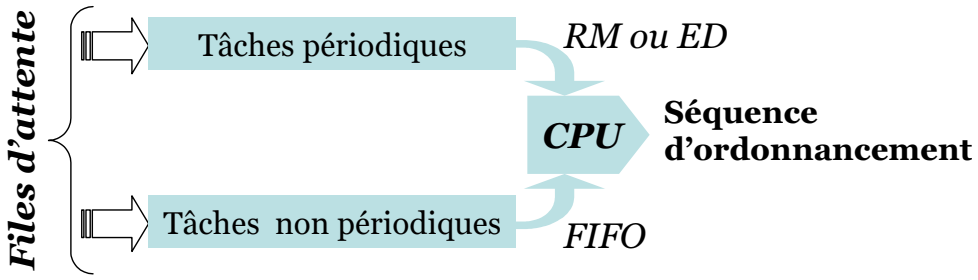

Fig. 2.8: Ordonnancement de tâches apériodiques par placement en arrière-plan (Background)

principaux, RM et ED. Dans la littérature, il n'est fait que très peu état des algorithmes DM et LL en présence d'ordonnancement conjoint de tâches apériodiques.
– La rigidité des contraintes temporelles des tâches apériodiques qui peuvent être relatives ou strictes.

2.3.5.1 Algorithmes de type serveur

Nous supposons qu'il y a n tâches périodiques, de charge totale U et une tâche serveur de charge U_s. L'ensemble de ces (n + 1) tâches est ordonnancé par RM. Par ailleurs, nous supposons que les requêtes apériodiques sont traitées dans l'ordre de leur arrivée, c'est à dire que la file d'attente des tâches apériodiques est gérée en mode FIFO.

Ce qui différencie les différents serveurs entre eux est la manière dont ils se comportent lorsqu'ils sont activés alors qu'il n'y a aucune requête apériodique en attente. Pour chacun d'eux, nous décrivons le principe de fonctionnement, puis nous donnons quelques résultats concernant l'analyse d'ordonnancabilité : il s'agit de déterminer, pour une charge de serveur donnée U_s la capacité périodique U maximale pour laquelle le respect des échéances des tâches périodiques sera garanti. Les critères dégagés sont les conditions suffisantes, et s'appuient sur le critère d'ordonnançabilité de [LL73] pour RM et ED.

Il est à noté que les serveurs utilisés sous ED ne font pas état de test de garantie des tâches sporadiques.

Serveur à scrutation

Le premier de ces algorithmes et le plus simple, *Serveur à Scrutation* ou *Polling Server*(PS), s'applique uniquement avec RM. Le flot d'arrivée des tâches apériodiques est stocké dans une file d'attente et exécuté durant la charge C_s de la tâche serveur. Lorsqu'il n'y a plus de tâches apériodiques actives (c'est à dire lorsque la file est vide), la capacité C_s est suspendue et ne sera réapprovisionnée qu'à la prochaine activation de la tâche serveur τ_s. Par conséquent comme nous le montre la figure 2.9, une tâche apériodique dont le réveil s'effectue juste après la suspension de la tâche serveur, doit attendre un minimum de $T_s - C_s$ unités de temps avant d'être exploitée.

Nous pouvons constater que cette méthode permet de stocker les apériodiques en attente puis de leur attribuer une priorité suivant un algorithme de type FIFO ou

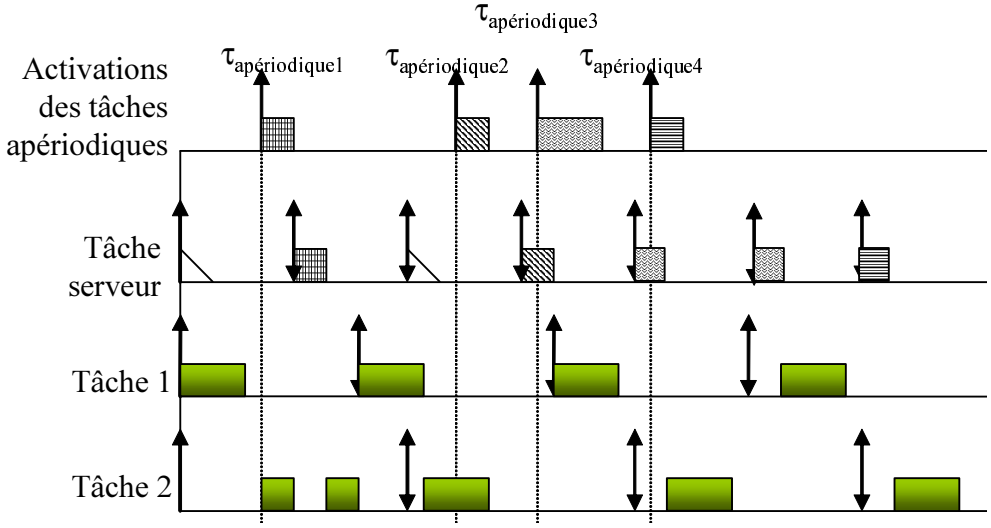

Fig. 2.9: Ordonnancement de tâches apériodiques par un serveur à scrutation dans un environnement de tâches périodiques ordonnancées sous RM. ($\tau_1 \langle 0, 4, 12, 12\rangle$, $\tau_2 \langle 0, 4, 16, 16\rangle$,$\tau_{serveur} \langle 0, 2, 8, 8\rangle$)

suivant un paramètre d'urgence pour les tâches critiques (comme une distribution de priorité).

Le pire cas en ce qui concerne l'ordonnancement des tâches périodiques se produit lorsqu'il y a en permanence des tâches apériodiques en attente, donc lorsque le serveur se comporte comme une tâche périodique. Par suite, le respect des échéances des tâches périodiques sera garanti par une CS

$$U_t = \sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C_s}{T_s} \leq (n+1)(2^{1/(n+1)} - 1)$$

Nous pouvons trouver dans [But97] une analyse permettant de mettre en évidence une CNS dans le cas où la tâche serveur est de priorité maximale pour les tâches sporadiques (contraintes temporelles strictes). Par ailleurs, en choisissant la plus grande période pour la tâche serveur nous augmentons de manière conséquente le temps de réponse moyen. Cet algorithme donne des résultats peu compétitifs pour le temps de réponse moyen des tâches apériodiques. En effet, la capacité du serveur est perdue dès lors qu'aucune tâche apériodique n'est active. Du coup, les activations de tâches apériodiques survenant juste après la suspension de la capacité et l'activation de la nouvelle instance du serveur sont pénalisées et obtiennent inévitablement de mauvais temps de réponse.

Serveur Ajournable

Pour améliorer la prise en compte des tâches apériodiques après leur activation et le temps moyen de réponse, Lehoczky, Sha et Strosnider [LSS87] ont introduit une amélioration au PS en supprimant le principe de suspension de la capacité. Ainsi

durant toute sa période, la tâche serveur est potentiellement capable de fournir un temps de traitement égal à sa capacité. De ce fait, une priorité maximale pour le serveur permet une meilleurs réactivité. Cette méthode a été nommée le *Serveur Ajournable* (appelée DS pour *Deferrable Server*).

Le principe de maintien de la capacité du serveur permet d'améliorer le temps de réponse moyen des tâches apériodiques mais engendre un bouleversement de l'ordonnancement des tâches périodiques. Or, ce bouleversement se traduit par une modification du test d'ordonnançabilité sous RM. En effet, l'ajournement d'une tâche prioritaire (en l'occurrence la tâche serveur s'il n'y a pas de requête apériodique) va à l'encontre des règles qui régissent Rate Monotonic à savoir que si la tâche de plus forte priorité est prête, elle doit s'exécuter et non pas revenir dans l'état 'en attente'. Il peut alors arriver qu'un dépassement d'échéance par une tâche périodique ait lieu uniquement parce que la tâche serveur ne s'est pas exécutée lorsqu'elle était prioritaire mais bien plus tard. Il existe toutefois une borne permettant de s'assurer que l'ordonnancement des tâches périodiques par RM sera valide :

$$U \leq n \left[\left(\frac{U_s + 2}{2U_s + 1} \right) - 1 \right]$$

Cette borne supérieure admet comme limite $L = Ln\left(\frac{Us+2}{2U_s+1}\right)$. Donc les tâches périodiques seront ordonnançables si $U < L$

Nous pouvons noter l'existence d'une version sous EDF de DS donnée par Ghazalie et Baker en 1995 [GB95] nommé Deadline Exchange Server. Ce dernier ne donne toutefois pas de résultats probants du point de vue du temps de réponse des tâches apériodiques.

Serveur à Échange de Priorité

Pour palier le problème de la perturbation du test d'ordonnançabilité des tâches périodiques, les mêmes auteurs ont fourni une autre méthode également basée sur le maintien de la capacité du serveur mais celle-ci perd de sa priorité au fur et à mesure qu'elle est ajournée. Cette méthode appelée Échange de Priorité ou Priority Exchange (PE) permet à une tâche périodique de priorité inférieure, d'échanger sa capacité avec celle de la tâche serveur pour permettre à la tâche périodique de s'exécuter avec la plus forte priorité, dans le cas où il n'y a pas d'apériodiques en attente. En cas d'échange préalable de capacité, si une tâche apériodique survient elle sera prise en compte seulement au niveau de priorité du dernier échange. Contrairement au PS, la capacité du serveur n'est pas perdue durant sa période, mais elle n'est pas non plus préservée au niveau de priorité de la tâche serveur comme pour DS. Le processus d'échange de priorité est propagé parmi les tâches de plus faible priorité, jusqu'à la dernière qui accumulera ainsi la capacité du serveur ou la dégradera en cas d'inactivité du processeur.

Les tests de garantie des tâches sporadiques sous PE sont difficiles à évaluer du fait

de la répartition de la capacité du serveur sur plusieurs niveaux de priorités. En effet, pour un ensemble de n tâches et dans le pire cas, la capacité est répartie sur n+1 niveaux de priorités. La vérification du respect du délai critique d'une tâche sporadique revient donc à chercher toutes les combinaisons sur les n+1 niveaux. Le serveur à échange de priorité fournit des temps de réponse un peu moins bons que PS (ceci étant dû à la baisse de priorité de la capacité du serveur en l'absence de tâches apériodiques). De plus, l'utilisation de PE est gourmande en mémoire puisque plus le nombre de tâches périodiques est grand, plus l'algorithme l'implémentant doit gérer les paramètres qui tiennent compte des échanges de capacités entre tâches.

Sous RM, l'avantage primordial que possède PE réside dans le fait qu'il ne perturbe pas le test d'ordonnançabilité des périodiques au contraire de DS. En effet, les seuls changements que ce protocoles engendre sont des exécutions anticipées des tâches périodiques mais jamais retardées. Il existe une CS portant sur l'ordonnancement des tâches périodiques par RM, si :

$$U < n\left[\left(\frac{2}{U_s+1}\right)^{1/n} - 1\right]$$

alors l'ordonnancement des tâches périodiques est valide. Cette borne supérieure admet comme limite $L = Ln(\frac{2}{U_s+1})$. Donc les tâches périodiques seront ordonnancables si U<L. Pour une charge serveur Us donnée, la charge périodique garantie est plus forte dans le cas du serveur à échange de prorités que dans le cas du serveur ajournable.

Sous ED, outre Deadline Defferable Server de [GB95] qui ne fournit pas de bon résultats en terme de temps de réponse, Spuri et Buttazzo [SB96a] ont proposé une version appelé *Serveur Dynamique à Échange de Priorité* aussi appelé DPE pour *Dynamic Priority Exchange Server*. Son intégration sous EDF modifie la technique d'échange qui ne tient plus compte d'une priorité basée sur la période des tâches, mais sur la proximité des échéances. Les performances de cet algorithme sous EDF demeurent satisfaisantes. [SB96a] ont permis de vérifier qu'un ensemble de tâches périodiques ayant un taux d'utilisation du processeur U_p et un algorithme à serveur DPE avec une charge processeur U_s, sont ordonnançable si et seulement si $U_p+U_s \leq 1$.

Serveur Sporadique

Il existe une variante au DS, qui maintient également la capacité du serveur, et dont les conditions d'ordonnançabilité sont analogues au cas de PS. Cette méthode appelée *Serveur Sporadique* (SS pour Sporadic Server) a été introduit par Sprunt, Sha et Lehoczky en 1989 [SSL89], et s'est posé comme objectif d'améliorer les performances de PE et DS pour les apériodiques à contraintes relatives mais également de pouvoir assurer les échéances des tâches à contraintes strictes.

La différence réside dans la façon de redistribuer la capacité après une consommation

par une tâche apériodique, ceci afin de ne pas remettre en cause l'ordonnancement des tâches périodiques, comme c'était le cas pour DS. Un serveur sporadique possède donc une capacité C_s et une période P_s qui ne sont utilisées qu'en cas d'activation d'une tâche apériodique au préalable. Dans ce cas, la prise en compte de l'apériodique ayant lieu au temps t, le réapprovisionnement de la capacité aura lieu au temps $t + T_s$, la quantité de réapprovisionnement dépend de ce qui a été consommé.

Cet algorithme possède une faible complexité d'implémentation qui peut être comparable à DS, et cela, en raison du maintien de la capacité du serveur à son niveau de priorité d'origine. De plus, cet algorithme permet une taille du serveur (définie par sa charge U_s) aussi bonne que dans le cas de PE (et meilleure que pour DS). Sous RM, même si SS viole les règles de fonctionnement de l'ordonnancement des tâches périodiques, [SSL89] ont montré qu'un ensemble de tâches périodiques ordonnançable l'est également si on remplace une des tâches périodiques par un serveur sporadique. La garantie des sporadiques est très difficile à obtenir formellement puisque la capacité du serveur est fragmentée en plusieurs morceaux de tailles différentes en accord avec les règles de réapprovisionnement. Par conséquent, calculer la date de terminaison d'une sporadique revient à pister toutes les recharges de capacités qui ont eu lieu durant le délai critique de la tâche. Toutefois, une solution est proposée par [SSL89] pour garantir les échéances des sporadiques, mais la méthode reste empirique. Celle-ci consiste à attribuer à chaque tâche à contraintes strictes, un serveur sporadique individuel.

Sous ED, [GB95] ont mis en évidence une adaptation simple de SS nommée Deadline Sporadic Server, mais ce dernier ne donne pas de résultat probant en ce qui concerne les temps de réponse. [SB96a] ont quand a eux proposé le *Serveur Dynamique Sporadique*(SDS). La différence qu'introduit le passage sous ED est bien entendu l'attribution dynamique de la priorité qui est choisie parmi une suite d'échéances dont la valeur est définie par la date future de réapprovisionnement de la capacité. Quand le serveur est créé, sa capacité est initialisée à sa valeur maximale Cs. La date de réapprovisionnement RT et l'échéance courante d_s du serveur sont définies aussitôt que $C_s > 0$ et qu'une requête apériodique est en attente. Soit t_a cette date, alors on a $RT = d_s = t_a + T_s$. La quantité RA devant être réapprovisionnée à la date RT est calculée quand la dernière tâche apériodique se termine ou quand C_s est épuisée. Soit t_i cette date, alors la valeur RA doit être égale à la capacité consommée entre l'intervalle $[t_a, t_i]$. Les possibilités de ce serveur s'avèrent intéressantes puisqu'il conserve les atouts de son homologue sous RM en terme de temps de réponse des tâche apériodiques. De plus, son implémentation reste simple et les coûts supplémentaires apportés au système très faibles.

Serveur à largeur de bande maximale

Malgré de bons résultats, le SDS peut être amélioré en partant du principe que l'utilisation d'un serveur de grande période peut retarder de manière significative l'exécution des tâches apériodiques. En effet, cette considération fait décroître les

chances d'être prioritaire par ED car l'échéance associée à l'arrivée d'une tâche apériodique sera décalée de T_s unités de temps. Or, comme T_s est considéré comme grand, il y a beaucoup de chance que des tâches périodiques soit prioritaires (dans le sens où leur échéance est la plus proche) devant ce serveur. Le *Serveur à largeur de bande maximale* (noté TBS pour *Total Bandwidth Server*) mis au point par [SB96b, SB94, CLB99, SBS95] contourne ce phénomène.

Intuitivement, l'approche la plus évidente est de faire en sorte de ne pas prendre de période trop grande pour le serveur. Le problème est qu'il n'est pas toujours possible, une fois la configuration de périodes déterminée de choisir un serveur de période inférieure pour des raisons d'ordonnançabilité. Une autre technique consiste à attribuer une échéance à chaque requête apériodique de manière à ne pas mettre en péril l'ensemble du système. Cette attribution d'échéance respecte le fait que l'utilisation du processeur par la charge apériodique ne dépasse jamais la valeur prédéfinie du serveur U_s. Rigoureusement, cette attribution s'effectue suivant la formule :

$$d_k = max(r_k, d_{k-1}) + \frac{c_k}{U_s}$$

où r_k est la date d'apparition de la k^{ime} tâche apériodique, d_k son échéance calculée, C_k sa charge et d_{k-1} l'échéance de l'apériodique précédent ($d_0 = 0$). Cette méthode reste simple à mettre en oeuvre et demeure l'une des plus performantes en terme de temps de réponse.

2.3.5.2 Algorithmes de type vol de temps creux

Une méthode simple pour la gestion des activations des tâches non périodiques consiste à utiliser une deuxième file de tâches et utiliser les temps d'inactivité issus de l'ordonnancement des tâches périodiques. Ainsi, le flux de tâches apériodiques est géré de façon transparente et suivant des règles simples comme FIFO. Mais, ce fonctionnement que l'on nomme méthode d'*arrière plan* ou *background* pouvant fonctionner autant avec ED et RM, souffre de mauvais résultats en terme de temps de réponse moyen des tâches apériodiques. Quant aux tâches sporadiques, rien ne peut être établi pour garantir le respect des contraintes temporelles. D'autres méthodes ont donc vu le jour pour palier ces problèmes.

Algorithme d'ordonnancement périodique à priorités fixes

Lehoczky et Ramos Thuel en 1992 [LRT92] ont mis au point une amélioration de la technique d'arrière plan que l'on nomme *Slack Stealing*. Le fonctionnement de cette méthode repose sur la constatation suivante : un ordonnancement conservatif (au plus tôt) des tâches périodiques n'est pas forcément indispensable. Nous ordonnançons a priori au plus tôt. Mais quand une requête apériodique survient, nous retardons les exécutions périodiques tout en préservant la propriété de validité pour servir au plus tôt les requêtes. L'idée est donc en présence de requêtes apériodiques de retarder certaines tâches périodiques sans pour autant mettre en péril le respect des échéances. Il s'agit d'intégrer non pas une tâche serveur comme

précédemment mais une tâche passive qui tente de collecter le maximum de temps creux pour le service des apériodiques. L'évaluation du nombre de temps creux et de leur position est effectuée par une analyse de la Laxité. La méthode bouleverse donc l'ordonnancement des tâches périodiques sous RM dès l'activation d'une tâche apériodique pour lui permettre d'obtenir le maximum de laxité disponible parmi toutes les tâches périodiques actives, pour son exécution. S'il n'y a pas d'apériodique à traiter, l'ordonnancement se poursuit normalement, selon les règles qui régissent RM. Cette prise de temps se traduit alors par une exécution au plus tard de la tâche périodique en cours. L'analyse d'ordonnançabilité fait appel à une fonction de laxité qui permet d'obtenir le maximum de temps creux sous RM, pour le service des apériodiques dans un intervalle de temps donné. Cette fonction permet la construction de tables stockant les dates de temps creux de façon pré-calculées [LRT92] ou dynamique [DTB93], stockées en mémoire et consultées en opérant une mise à jour suivant la configuration des périodiques, du service des apériodiques et des temps creux. Le calcul de la laxité s'opère de manière statique et se calcule en complexité O(n) où n représente le nombre de tâches périodiques. Malheureusement pour cet algorithme, le stockage des tables (qui s'accroît considérablement avec une grande métapériode) nécessaire au calcul de laxité, empêche toute possibilité viable d'implémentation. Le traitement des tâches sporadiques nécessite également un test de garantie des contraintes temporelles qui repose sur les travaux de [CC89]. Nous pouvons noter l'existence d'autres méthodes comme l'Echange de Priorité Étendu (EPE) basé sur l'approche serveur PE mais utilisant les temps d'inactivité processeur issus de l'évaluation pire cas des durées d'exécution des tâches périodiques. Ce surplus de temps processeur est alors directement exploité pour le traitement des tâches non périodiques.

Algorithme d'ordonnancement périodique à priorité dynamique

Les travaux de [CC89] sur l'ordonnancement sous ED ont permis de mettre en évidence une méthode de traitement des tâches non périodiques en tirant profit de deux techniques d'ordonnancement issues de ED : EDF et EDL qui exécutent respectivement au plus tôt et au plus tard les tâches concurrentes. Cette méthodes appelé par la suite *Serveur EDL* par [SB96b] consiste à ordonnancer les tâches périodiques, quand il n'y a pas de requêtes apériodiques, suivant EDF pour permettre de garder un maximum de temps creux pour la fin de la séquence établie sur la métapériode. Lors d'une requête d'une tâche apériodique, l'ordonnancement passe en EDL (en se référant aux données pré-calculées) et permet ainsi de maximiser les temps creux au moment de la requête. Bien entendu, le fait d'appliquer EDL implique un calcul dynamique d'ordonnancement des tâches périodiques actives au moment de l'arrivée de la tâche apériodique. Cette méthode permet d'avoir un maximum de temps creux disponibles dès la requête apériodique. Cette méthode a été montrée comme étant optimale vis à vis du temps de réponse moyen obtenu pour les tâches apériodiques. Cet algorithme est en définitif très proche du Slack Stealer. Nous pouvons trouver dans [CC89, SCE90] une approche permettant la gestion des tâches sporadiques grâce à l'utilisation de fonctions de garantie permettant d'évaluer si la tâche peut

être acceptée dynamiquement dans le système. [CD93, Sil94] proposent des simulations de cette méthode rendant compte de ses performances. Les résultats sur EDL et EDF ont également permis de proposer une adaptation du DPE, nommé *Serveur à Échange de Priorité Amélioré* ou *Improved Exchange Algorithm (IPE)* pouvant être ainsi utilisée sur ED. Les résultats obtenus sont assez proches de ceux du serveur EDL mais nécessitent également un pré-calcul des positions et des durées des temps creux ainsi qu'une réévaluation dynamique en cours d'exécution ce qui est fortement coûteux en temps processeur et en espace.

Nous pouvons constater que tous les algorithmes présentés ne permettent pas en général de garantir aisément les tâches sporadiques qui sont pourtant souvent essentielles dans un système temps réel. Nous pouvons trouver dans [SS94] une analyse permettant de prendre en compte aussi bien les problèmes de ressources que de contraintes de précédences. [SCE90] ont analysé les possibilités offertes par EDF pour l'ordonnancement des apériodiques sous contraintes de précédence.

2.4 Résultats de l'approche Hors-Ligne

Dès lors que les tâches d'un système temps réel partagent des ressources et sont liées par des contraintes de précédence, les méthodes En-Ligne montrent leur faiblesse. En effet, il n'existe aucun algorithme d'ordonnancement polynomial optimal dans ce cas général. Les stratégie d'ordonnancement Hors-Ligne ont été développées pour étudier les applications très contraintes que les ordonnancements En-Ligne ne savent pas traiter. Ce sont des méthodes qui reposent le plus souvent sur une approche modèle de l'ordonnancement qui ont une puissance d'ordonnancement optimale (elles savent détecter les ordonnancements valides s'il en existe) mais de complexité souvent élevée car elles nécessitent la construction et l'analyse de l'ensemble des solutions possibles. L'ordonnancement hors ligne consiste à déterminer avant la mise en marche du système l'enchaînement des tâches à exécuter sur le processeur. Un *plan* (ou séquence statique) est ainsi calculé et implémenté directement dans le noyau temps réel. L'ordonnanceur ne gère alors que le temps (et les problèmes de synchronisation des horloges locales réparties sur le système) pour un bon déroulement de la séquence.

Or, définir un plan nécessite de connaître la fin de l'exécution de l'application temps réel. Nous avons considéré qu'une application temps réel était composée de tâches périodiques. Ce type d'application fonctionne par conséquent en régime permanent puisque chaque tâche est activée périodiquement et ce, potentiellement à l'infini. Afin de valider un ordonnancement de tâches, et donc construire une séquence il faut se ramener à une durée d'étude bornée.Une première étude, dans le cas spécifique de ED, pour des tâches indépendantes, a été réalisée par [LM80, LW82] et a fourni une borne de la durée de la séquence à analyser. Puis [CGG04] ont donné la durée minimale de la séquence à construire, et ce pour toute configuration, et tout algorithme (déterministe).

Théorème 11 ***Durée d'étude 1***

Une configuration de n tâches, à échéances inférieures ou égales aux périodes et à départs simultanés, est ordonnançable par un algorithme déterministe si et seulement si la séquence produite est valide entre 0 et la Metapériode $P = Ppcm(T_i)$.[a]

[a] P est donc le Plus Petit Commun Multiple des périodes des n tâches.

D'autres résultats ont été mis en évidence pour un algorithme d'ordonnancement particulier : Earliest Deadline. [LM80] ont ainsi étendu le résultat précédent au cas des tâches à départs différés.

Théorème 12 ***[LM80] Durée d'étude 2***

Une configuration de n tâches indépendantes, à échéances inférieures ou égales aux périodes, est ordonnançable par Earliest Deadline si et seulement si la séquence d'ordonnancement produite est valide sur l'intervalle

$$[0, max_{i=1,\ldots,n}\{r_i\} + 2P]$$

[LM80] constate alors que l'état du système est identique aux temps $max_{i=1,\ldots,n}\{r_i\} + 2P$ *et* $max_{i=1,\ldots,n}\{r_i\} + P$. Cette dernière étude a été reprise par [CGG04] et a permis d'obtenir une généralisation à toutes les configurations de tâches dans un contexte monoprocesseur et pour tout algorithme d'ordonnancement. Ce résultat est basé sur l'analyse des temps creux. Ces derniers sont scindés en deux catégories, les temps creux cycliques et le temps creux acycliques. Les premiers font partie du régime permanent de l'ordonnancement alors que les temps creux acycliques sont en nombre borné et dus exclusivement à la montée en charge du processeur et aux dates de premières activations de chaque tâche (paramètre r_i). Ainsi une configuration de tâches à départs différés ordonnancée par tout algorithme conservatif présentera n_{ac} temps creux acycliques situés en début de séquence. Nous notons t_c la date d'apparition du dernier temps creux acyclique. Á partir de cette date, tous les temps creux sont cycliques. De cette propriété, [CGG04] ont déduit une durée d'étude optimale :

Propriété 3 ***[CGG04, Gro99] Durée d'étude 3***

Un ordonnancement d'un système de tâches de charge $U \leq 1$ *est valide si et seulement si il est valide sur l'intervalle*

$$I = [0 \ldots t_c + P + 1]$$

ou t_c *est la date d'apparition du dernier temps creux acyclique.*

De plus, le dernier temps creux acyclique ne peux apparaître au delà de la date

$max_{i=1...n}(r_i)+P$ ce qui permet de vérifier le résultat de [LM80] puisque si $t_c = max_{i=1...n}(r_i)+P-1$ alors l'intervalle I devient $[0 \ldots max_{i=1...n}(r_i) + 2P]$ (Définition 12).

La figure 2.10 met en évidence la date de dernier temps creux acyclique permettant de réduire de façon optimale la durée d'étude d'ordonnançabilité d'une application temps réel.

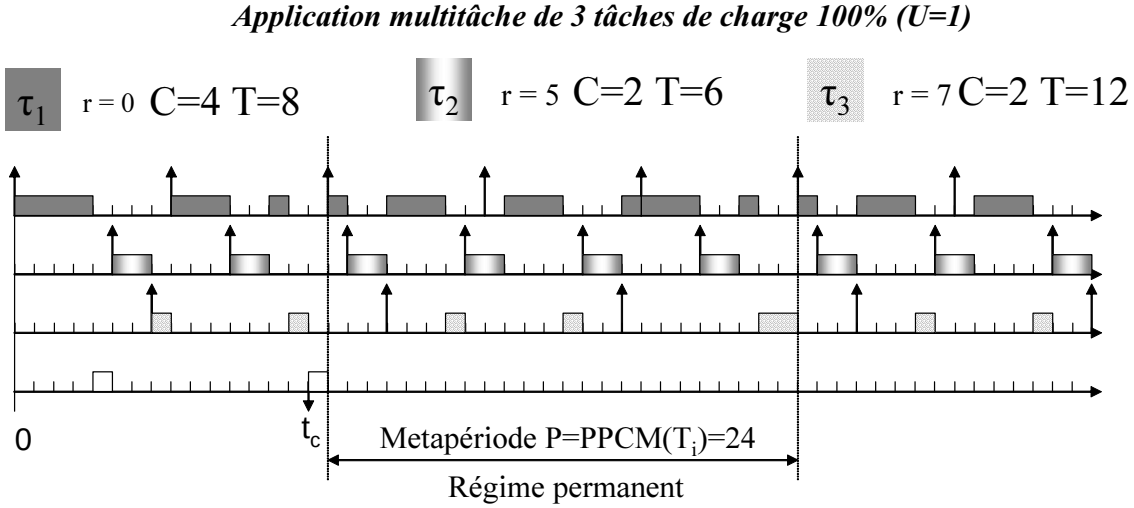

Fig. 2.10: Application temps réel de charge 100% ($U = \frac{4}{8} + \frac{2}{6} + \frac{2}{12} = 1$) composée de trois tâches, ordonnancée par un algorithme conservatif. Il apparait deux temps creux acycliques au temps 4 et 15. Le régime permanent démarre donc au temps 16. On constate que les tâches sont dans le même état au temps 16 et une métapériode plus tard à savoir au temps 40. La durée d'étude suffisante à la validation correspond donc à l'intervalle [0,40].

L'optimalité de la durée d'étude permet ainsi de gagner beaucoup en temps de calcul sur les méthodes d'ordonnancement Hors-Ligne puisque les séquences à étudier sont réduites à leur taille minimale.

L'atout majeur des méthodes Hors-Ligne reste la rigueur et la robustesse. En effet, aucun changement ne peut être appliqué en cours de route ce qui supprime les anomalies d'ordonnancement. De plus, la phase de validation nécessaire avec les méthodes En-Ligne est ici inutile puisque seules les séquences valides sont construites. La séquence produite est figée dans le temps et n'est ainsi pas assujettie aux variations de l'environnement ni aux variations du temps d'exécution des tâches d'une instance à une autre.

Toutefois, ces avantages ont un prix, d'un coût exponentiel pour l'élaboration d'une solution optimale. Et la rigueur constatée peut également devenir un désavantage en terme de rigidité pour les éventuelles possibilités d'évolution dynamique du système. De plus, pour élaborer une séquence d'ordonnancement hors-ligne, tous les paramètres temporels doivent être déterministes (ce qui interdit les tâches apériodiques et sporadiques).

Nous pouvons différencier deux approches pour l'étude hors-ligne de séquence d'ordonnancement :

– soit une analyse exhaustive des solutions par des méthodes d'optimisation exactes comme le Branch and Bound par exemple qui est utilisé dans les travaux de [Ric02, XP90, XP92, BS74], ou encore par une modélisation par *automates finis* [LCG04,

LG02], par modèle géométrique [LG05] ou par *Réseau de Petri* [GCG02] des systèmes dont la finalité est soit construire une solution pour tester l'ordonnançabilité de l'application soit parcourir l'ensemble des solutions d'ordonnancement pour choisir une séquence valide optimale.

– soit par des méthodes approchées où l'on perd l'optimalité de la solution trouvée mais au bénéfice d'un gain en temps de calcul. Ces techniques utilisent des algorithmes génétiques, de recuit simulé, des fourmis *etc...*[JP96].

2.5 Bibliographie

[ABD+95] N.C. Audsley, A. Burns, R.I. Davis, K.W. Tindell, and A.J. Wellings. Fixed priority pre-emptive scheduling : an historical perspective. *Real-Time Systems*, 8 :173–198, 1995.

[ABRW91] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard real-time scheduling : the deadline monotonic approach. *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 127–132, 1991.

[ABRW92] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Deadline monotonic scheduling theory. *Real-time Programming*, pages 55–60, 23-26 juin 1992.

[Aud91] N.C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. *Technical Report YCS-164*, 31, nov 1991.

[Bak91] T.P. Baker. Stack-based scheduling of real-time processes. *the Journal of Real-Time Systems*, 3 :67–99, 1991.

[BG04] Sanjoy Baruah and Joel Goossens. *Scheduling Real-time Tasks : Algorithms and Complexity. In Handbook of Scheduling : Algorithms, Models, and Performance Analysis, Joseph Y-T Leung.* Chapman Hall/ CRC Press, 2004.

[BHR90] S. Baruah, R. Howel, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic real time tasks on one processor. *Real time systems*, 2 :301–324, 1990.

[Bla76] J. Blazewicz. Scheduling dependant tasks with different arrival times to meet deadlines. *Modeling and performance evaluation of computer systems*, pages 57–65, 1976.

[BMR90] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. *Proc. 11th IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[BS74] K.R. Baker and Z.S Su. Sequencing with due-dates and early start times to minimize maximum tardiness. In *Naval Research Logistic Quaterly*, volume 21, pages 171–176, 1974.

[But97] G.C. Buttazzo. *Hard real time computing systems : predictable scheduling algorithms and applications.* Kluwer Academic Publisher, 1997.

[CC89] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE TRANS. ON SOFT. ENGENEERING*, 15(10) :1261–1269, 1989.

[CD93] H. Chetto and J. Delacroix. Minimisation des temps de réponse des tâches sporadiques en présence des tâches périodiques. *Salon des solutions informatiques temps réel*, pages V39–V51, January 1993.

[CDKM00] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Oronnancement Temps Réel*. Hermès Edition, 2000.

[CET01] Samarjit Chakraborty, Thomas Erlebach, and Lothar Thiele. On the complexity of scheduling conditional real-time code. *Proceedings of the Seventh International Workshop on Algorithms and Data Structures (WADS 2001)LNCS 2125*, pages 38–49, 2001.

[CGG04] A. Choquet-Geniet and E. Grolleau. Minimal schedulability interval for real time systems of periodic tasks with offsets. In *Theoretical of Computer Sciences*, volume 310, pages 117–134, March 2004.

[Che02] Albert M. K. Cheng. *Real Time Systems, Scheduling Analysis and Verification*. John Wiley & Sons, 2002.

[CL90] M. Chen and K. Lin. Dynamic priority ceilings : a concurrency protocol for real time systems. *Real time systems*, no. 2 :235–346, 1990.

[CLB99] M. Caccamo, G. Lipari, and G. Buttazzo. Sharing ressources among periodic and aperiodic taskc with dynamic deadlines. *In proceeding of th* 20^{th} *IEEE Real Time System Symposium*, 1999.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. *In proceeding of the third annual ACM symposium on Theory of Computing*, 1971.

[Coo83] Stephen A. Cook. An overview of computational complexity. *Communication of ACM*, 26 :400–408, 1983.

[Cot99] F. Cottet. Etude de l'application temps réel embarquée : mission pathfinder. *actes de l'école d'été temps réel, ETR99*, pages 109–117, 13-16 sept 1999.

[Der74] M. Dertouzos. Control robotics : the procedural control of physical processors. *Proc. IFIP Congress*, pages 807–813, 1974.

[DM89] M.L. Dertouzos and A.K. Mok. Multiprocessor on-line scheduling of hard real-time tasks. *IEEE Transactions on Software Engineering*, 15(12) :1497–1506, Dec. 1989.

[DTB93] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. *In proceeding of IEEE Real Time System Symposium*, pages 22–231, 1993.

[GB95] T.M. Ghazalie and Theodore P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9(1) :31–67, 1995.

[GCG02] Emmanuel Grolleau and Annie Choquet-Geniet. Off-line computation of real-time schedules using petri nets. In *Discrete Event Dynamic Systems, DEDS*, volume 12(3), pages 311–333. Kluwer Academic Publishers, Jully 2002.

[Gen00] Annie Geniet. *Systèmes paralléles et temps réel : analyse à l'aide de modèles formels*. PhD thesis, Habilitation a diriger des recherches, LISI/ENSMA, Decembre 2000.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractibility : a guide to the theory of NP-completness*. W H Freeman & Co, 1979.

[Gra69] R. Graham. Bounds on multiprocessing timing anomalies. In *SIAM Journal of Applied Mathematics 17, no. 2*, pages 416–429, 1969.

[Gro99] E. Grolleau. *Ordonnancement temps-réel hors-ligne optimal à l'aide de réseaux de Petri en environnement mono-processeur et multiprocesseur*. PhD thesis, ENSMA - Université de Poitiers, November 1999.

[Har87] P.K. Harter. Response time in level structured systems. *ACM Transactions on Computer Systems*, 5(3) :232–248, 1987.

[Hen75] R Henn. *Deterministic modelle fur die prozessorzuteilung in einer harten realzeit-umgebung*. PhD thesis, Technical university, Munich, 1975.

[Jac55] J.R. Jackson. Scheduling a production line to minimize maximum tardiness. *Management Sciences Research Project*, 1955.

[Jos85] M. Joseph. On a problem in real-time computing. *Information Processing Letters*, 20(4) :173–177, 1985.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *Computer Journal*, 29(5) :390–395, 1986.

[JP96] Beauvais. J.-P. *Etude d'algorithmes de placement de tâches temps réel périodiques complexes dans un système réparti*. PhD thesis, Thèse de doctorat de l'Ecole Centrale de Nantes et de l'Université de Nantes., 1996.

[JSM91] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *In Proceedings of the 12 th IEEE Symposium on Real-Time Systems*, pages 129–139, Decembre 1991.

[Kai82] C. Kaiser. Exclusion mutuelle et ordonnancement par priorité. *Technique et Science Informatiques*, 1982.

[Kop97] Hermann Kopetz. *Real-Time Systems : Design Principles for Distributed Embedded Applications. Series : The International Series in Engineering and Computer Science, Vol. 395*, volume ISBN : 0 7923 9894 7. Kluwer Academic Publishers, 1997.

[KSSK96] H. Kaneko, J.A Stankovic, S. Sen, and K.Ramamritham. Integrated scheduling of multimedia and hard real time tasks. *Technical report UMCS 1996-045*, Computer Science, 1996.

[Lab74] J. Labetoulle. Un algorithme optimal pour la gestion des processus en temps réel. *Revue Française d'Automatique, Informatique et Recherche Opérationnelle*, pages 11–17, 1974.

[LCG04] Gaëlle Largeteau, Bernard Chauvière, and Dominique Geniet. Une approche géométrique de la validation d'applications temps réel. In *Real-Time Systems*, 2004.

[Leh90] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. *Proc. IEEE Real-Time Systems Symposium*, pages 201–209, 1990.

[LG02] Gaëlle Largeteau and Dominique Geniet. Validation temporelle d'applications temps-réel distribuées à contraintes strictes. In *Real-Time Systems*, 2002.

[LG05] Gaëlle Largeteau and Dominique Geniet. Discrete geometry applied in hard real-time systems validation. *Proc. of 12th Discrete Geometry for Computer Imagery, Lecture Notes in Computer Science*, 3429 :23–33, Springer-Verlag 2005.

[Liu00] J. Liu. *Real-time systems.* Prentice Hall, 2000.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for mutltiprogramming in real-time environnement. *Journal of the ACM*, 20(1) :46–61, 1973.

[LM80] J.Y.T. Leung and M.L. Merill. A note on preemptive scheduling of periodic real-time tasks. In *Information Processing Letters 11(3)*, pages 115–118, November 1980.

[LRT92] J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft aperiodic tasks in fixed priority preemptive system. *Proceeding of the Real Time System Symposium, IEEE*, pages 110–123, december 1992.

[LSD89] J.P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm : exact characterization and average case behavior. *Proc. 10th Real-Time Systems Symposium*, pages 166–171, december 1989.

[LSS87] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced periodic responsiveness in hard real time environments. In *Proceeding of the RTSS, IEEE*, pages 261–270, San Jose, December 1987.

[LSST91] J.P. Lehoczky, L. Sha, J.K. Strosnider, and H. Tokuda. Fixed priority scheduling theory for hard real-time systems. *in Foundations of Real-Time Computing : Scheduling and resource management*, pages 1–30, 1991.

[LW82] J. Leung and J. Whitehead. On the complexity of fixed priority scheduling of périodic real-time tasks. *Performance Evaluation*, pages 237–250, 1982.

[MA98] Y. Manabe and S. Aoyagi. A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling. *Real-Time Systems*, 14(2) :171–181, Mars 1998.

[Mok83] A.K. Mok. *Fundamental design problems for the hard real time environments.* PhD thesis, MIT, 1983.

[RC99] P. Richard and F. Cottet. Ordonnancement temps réel monoprocesseur avec contraintes de précédence simples et généralisées. *Tech. Report 99-002*, 1999.

[RCM96] I. Ripoll, A. Crespo, and A.K. Mok. Improvement in feasibility testing for real-time tasks. *Journal of Real-Time Systems*, pages 19–39, 1996.

[RCR01] P. Richard, F. Cottet, and M. Richard. Online scheduling of real time distributed computers with complex communication constraints. 7^{th} *International Conference on Engineering of Complex Computer Systems*, pages 23–24, 2001.

[Ric02] Michaël Richard. *Contribution à la Validation des Systèmes Temps Réel Distribués : Ordonnancement à Priorités Fixes & Placement.* PhD thesis, LISI, ENSMA, Université de Poitiers, Novembre 2002.

[Ric03] Pascal Richard. Analyse du temps de réponse des systèmes temps réel. *Actes de l'École d'Été Temps Réel*, pages 241–262, 2003.

[SB94] M. Spuri and G Buttazzo. Efficient aperiodic service under earliest deadline scheduling. *In proceeding of th* 15^{th} *IEEE Real Time System Symposium*, pages 2–21, 1994.

[SB96a] M. Spuri and G Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *the Journal of Real-Time Systems*, 10 :179–210, 1996.

[SB96b] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *The Journal of real time systems*, 10 :179–210, 1996.

[SBS95] M. Spuri, G. Buttazzo, and F. Sensini. Robust aperiodic sceduling under dynamic priority systems. *In proceeding of IEEE Real Time System Symposium*, december 1995.

[SCE90] M. Silly, H. Chetto, and N. Elyounsi. An optimal algorithm for guaranteeing sporadic tasks in hard real time systems. *IEEE Symposium on Parallel and Distribued systems*, pages 578–585, Dec 1990.

[Ser72] O. Serlin. Scheduling of time critical processes. *proc. Spring Joint Computers Conference*, pages 925–932, 1972.

[Sil94] M. Silly. Un algorithme d'ordonnancement des tâches sporadiques pour les systemes temps réel. *APII,*, 28, nř2 :179–205, 1994.

[SRL87] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inherence protocols. *Tech. report CMU-CS-87-181*, December 1987.

[SRL90] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inherence protocols : An approach to real time synchronization. *IEEE Transactions on Computers*, 39 :175–185, December 1990.

[SS93] L. Sha and S.S. Sathaye. A systematic approach to designing distributed real-time systems. *IEEE Computer*, 26 :68–78, sept 1993.

[SS94] M. Spuri and J. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. *IEEE Transactions on computer*, 12 :1407–1412, 1994.

[SSDB94] J.A. Stankovic, M. Spuri, M. DiNatale, and G. Buttazzo. Implications of classical scheduling results for real time systems. *IEEE computer*, vol 28-N6, 1994.

[SSL89] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic task scheduling for hard real time systems. *The journal of Real Time Systems*, 1 :27–60, 1989.

[Tin76] K.W. Tindell. *Fixed priority scheduling of hard real time systems.* PhD thesis, University of York, 1976.

[TLS96] T.S. Tia, J.W.S Liu, and M. Shankar. Algorithms and optimality of scheduling soft aperiodique requests in fixed priority preemptive systems. *The journal of Real Time Systems*, 10, nř1 :23–43, 1996.

[XP90] J. Xu and D. Parnas. Scheduling processes with releases times, deadlines precedence and exclusion relations. In *IEEE transactions on Software Engineering*, volume 16(3), pages 360–369, march 1990.

[XP92] J. Xu and D. Parnas. Pre-runtime scheduling of processes with exclusion relations on nested or overlapping critical sections. In *Phoenix Conference on Computers and Communications*, pages 6.4.7.1–6.4.7.9, April 1992.

[ZS94] Q. Zheng and K.G. Shin. On the ability of establishing real-time channels in point-to-point packet-switched networks. *IEEE Transactions on Communications*, pages 42(2,3,4), 1994.

Chapitre Trois

Modélisation d'un Système Temps Réel par Réseau de Petri

Chapitre Trois

Modélisation d'un Système Temps Réel par Réseau de Petri

Les Réseaux de Petri constituent un outil puissant et intuitif de modélisation de processus. Nous allons dans un premier temps introduire brièvement les réseaux de Petri et plus particulièrement les extensions permettant de prendre en compte le temps. Nous montrons alors comment analyser hors ligne une application temps réel constituée de tâches périodiques fortement couplées à l'aide d'un réseau de Petri. Tout d'abord, nous modélisons l'application, puis nous construisons un graphe des marquages adapté, duquel nous pourrons extraire des séquences d'exécution pertinentes.

Sommaire

Chapitre Trois

Modélisation d'un Système Temps Réel par Réseau de Petri

Le mode de fonctionnement des applications de contrôle de procédé doit être validé lors de la phase de conception de l'application. La phase de validation pour être fiable, doit s'appuyer sur des méthodes formelles qui proposent des approches rigoureuses et sont dotées d'outils de preuve. Plusieurs approches peuvent être envisagées :

- L'approche orientée propriété, où les systèmes sont modélisés par des langages. Dans ce cas, valider le fonctionnement du système revient à prouver certaines propriétés de ce langage. Nous pouvons pour cela utiliser par exemple la logique ou encore la logique temporelle.
- L'approche orientée modèle qui permet de décrire les systèmes ainsi que leurs actions. L'objectif est de fournir un outil de simulation du fonctionnement du système considéré. De nombreux modèles ont été définis, citons à titre d'exemple les systèmes de transitions, les automates finis, les statecharts, *etc.* ...

Nous nous intéressons par la suite à cette dernière approche. Nous avons choisi le modèle des *réseaux de Petri* [Pet62, HSSM68, Pet80] (que l'on notera par la suite RdP) qui a initialement été conçu pour modéliser les processus communicants et les structures de contrôle. Les applications concurrentes et plus précisément pour ce qui nous intéresse, les applications temps réel présentées dans les chapitres précédents s'inscrivent pleinement dans ce contexte. Notre objectif global est de déterminer un mode de fonctionnement de l'application dont nous pourrons garantir la validité. Pour ce qui concerne le fonctionnement, nous focalisons notre attention sur les aspects ordonnancement ; quant au critère de validité retenu, il correspond au respect des contraintes temporelles. Nous avons vu précédemment que les techniques d'ordonnancement en ligne arrivaient à leurs limites lorsque nous considérons des applications utilisant des ressources critiques et des synchronisations. Notre approche s'inscrit dans le cadre général de l'ordonnancement hors ligne des applications. Il s'agit de déterminer un comportement valide qui sera ensuite celui qu'adoptera l'application. Le modèle des RdP permet de modéliser de manière fine les applications comportant des tâches interagissantes et de décrire pas à pas les évolutions possibles et plus précisément les comportements d'exécution. Il semblait donc particulièrement bien adapté à notre problématique, ce qui explique notre choix. Nous nous sommes appuyés sur une modélisation pré-existante, proposée par [CGC96] puis [GCG02] que nous avons étendue et adaptée à nos besoins.

Dans un premier temps, nous introduisons le modèle Places-Transitions, modèle de base des RdP. Puis, pour pouvoir tenir compte du caractère primordial des systèmes

temps réel, à savoir la prise en compte du temps, nous dressons un bref panorama des extensions temporelles des RdP [CGVN93, Pet81, Mur89, Jen96, BF86, CG06]. Parmi celles-ci, les RdP fonctionnant avec la règle de tir maximal ont été retenus. Dans un deuxième temps, nous présentons en détail la modélisation de [GCG02] des systèmes temps réel comportant uniquement des tâches périodiques à durées déterministes et les techniques de détermination d'une séquence d'ordonnancement valide.

3.1 Les Réseaux de Petri

3.1.1 Définition des Réseaux de Petri autonomes

Les réseaux de Petri permettent d'étudier des systèmes dynamiques complexes. Ils ont été proposés en 1962 par Carl Adam PETRI [Pet62], puis développés au MIT. Ils sont maintenant utilisés pour spécifier, modéliser et comprendre les systèmes (au sens informatique) dans lesquels plusieurs processus sont interdépendants. Ils constituent un outil graphique et mathématique de modélisation. Nous adoptons pour la suite les notations utilisées dans [BF86] et repries par [Gro99].

Définition 11 ***Définition des Réseaux de Petri***

Un RdP marqué *est défini par* $R = \langle N, M_0 \rangle$ *où :*

- N *est un réseau Place-Transition* $\langle P, T, W \rangle$ *avec :*
  - $P = \{p_1, \ldots, p_n\}$ *est un ensemble fini de* ***places*** ,
  - $T = \{t_1, \ldots, t_m\}$ *est un ensemble fini de* ***transitions*** *avec* $P \cap T = \oslash$,
  - $W : (P \times T) \cup (T \times P) \longrightarrow \mathbb{N}$ *est la fonction de valuation,*
- $M_0 : P \longrightarrow \mathbb{N}$ *définit l'état initial de tous les éléments de P, c'est le marquage initial.*

Concrètement, on représente un RdP par un graphe bipartite : les places sont représentées par des ronds et les transitions par des rectangles. De plus, le marquage initial est représenté par des jetons dans les places et se définit comme un vecteur donnant le nombre de jetons que contient chacune des places. La figure 3.1 montre la représentation d'un RdP à 2 places, 1 transition et 2 arcs de pondérations respectives 1 et 2. Intuitivement, les places (et les marquages) permettent de décrire l'état (instantané) du système, les transitions correspondent aux actions que le système peut effectuer et les arcs expriment les conditions requises pour qu'une action puisse être exécutée ainsi que l'effet de cette action sur l'état du système.

L'évolution du système est modélisée par le tir d'une transition. Nous disposons par conséquent, en plus d'une représentation statique des états du système, d'une modélisation dynamique de son évolution. Une action est exécutable, ce qui revient à savoir si la transition associée à l'action est franchissable, si les conditions spécifiées par les arcs pointant vers cette transition sont vérifiées :

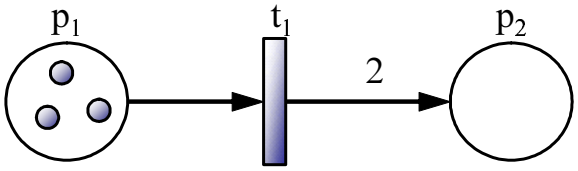

FIG. 3.1: Exemple d'un RdP possédant 2 places (p_1 et p_2) et 1 transition t_1. L'arc de P_1 à t_1 est valué par 1 (par convention, on omet volontairement les valuations dans le cas unitaire) et une pondération de 2 sur le deuxième arc. Le marquage initial M_0 est de 3 jetons dans la place p_1 et aucun dans la place p_2. À partir de l'état M_0 la transition t_1 peut être franchie trois fois successivement ce qui produit 2 jetons à chaque franchissement dans la place p_2. Ce processus restera donc bloqué avec un marquage final de 0 jeton dans p_1 et 6 dans p_2.

DÉFINITION 12 ***Franchissement des transitions***

Soit M le marquage courant du réseau de Petri et $M(p)$ le marquage de la place p alors :

- *Une transition $t \in T$ est* ***franchissable*** *à partir d'un marquage M si et seulement si $\forall p \in P,\ M(p) \geq W(p,t)$.*
- *Le franchissement de la transition t consomme $W(p,t)$ jetons et en produit $W(t,p)$ dans toute place p, conduisant au marquage M' défini par $\forall p \in P,\ M'(p) = M(p) - W(p,t) + W(t,p)$.*
- *Le franchissement de t à partir du marquage M mène au marquage M', et on note $M(t > M'$.*

Nous étendons ensuite les règles de franchissement d'une transition à une séquence de transitions. Nous passons ainsi de l'exécution d'une action à celle d'une séquence d'actions. Une séquence de transitions ou de franchissements w est une suite de transitions $< t_1, t_2, \ldots, t_n >$ qui permet, à partir d'un marquage M, de passer au marquage M' par le franchissement successif des transitions définissant la séquence. On le note $M(t_1 \cdot t_2 \ldots t_n > M'$.

Un RdP est défini de façon statique par sa structure et son état initial (M_0). Sa dynamique est donnée par la règle de tir (ou franchissement), lui permettant d'évoluer de marquage en marquage. L'ensemble des marquages atteignables à partir de l'état initial M_0 permet de décrire l'ensemble des états que le système peut prendre.

L'ensemble des séquences franchissables à partir du marquage initial est appelé le *langage* du réseau. Il décrit l'ensemble de tous les comportements possibles du système.

Si nous associons une lettre d'un alphabet A à toute transition du réseau, ce qui peut être vu comme un morphisme h de A dans T, nous définissons un RdP étiqueté (N,h). L'image par h du langage est le langage du réseau étiqueté : dans chacune des séquences, nous avons remplacé chaque transition par sa lettre étiquette.

Définition 13 ***Langage et*** **Marquage Accessible**

Soit le RdP étiqueté (N,h) et le marquage initial M_0 alors :
– *Le* ***langage*** *de N est défini par* $L = \{w \in T^* / M_0(w >\}$,
– *Le* ***langage étiqueté*** *est définit par* $L_R = \{h(w)/w \in L\}$
– *L'ensemble des* ***marquages accessibles*** *noté* $Acc(N, M_0)$ *est défini par :*

$$Acc(N, M_0) = \left\{M \in \mathbb{N}^{|P|} / \exists w \in L / M_0(w > M\right\}$$

Lorsque le nombre de marquages accessibles est fini, ce qui se traduit par le fait que toutes les places sont bornées (une place est bornée si pour tout marquage accessible, le nombre de jetons qu'elle contient est borné), nous parlons de *RdP borné*.

L'ensemble $Acc\,(N, M_0)$ peut être représenté sous la forme d'un graphe, fini si le RdP est borné, appelé *graphe des marquages* : chaque sommet est étiqueté par un marquage du réseau, et il existe un arc étiqueté par une transition t d'un sommet d'étiquette M à un sommet d'étiquette M' si et seulement si $M(t > M'$. Ainsi, l'ensemble des étiquettes des sommets du graphe représente exactement l'ensemble des marquages accessibles $Acc\,(N, M_0)$ et l'ensemble des étiquettes des chemins du graphe issus du sommet d'étiquette M_0 correspond au langage du réseau.

Le langage fournit sans distinction tous les comportements possibles du système modélisé. Il peut être utile de ne sélectionner que certains d'entre eux. Par exemple, nous pouvons ne considérer que les comportements amenant le système dans des états vérifiant certaines conditions, ou bien encore aboutissant à un état spécifique du système considéré comme état de terminaison. Pour cela, nous introduisons l'ensemble ξ appelé ensemble des *marquages terminaux* définissant les marquages à atteindre. Cet ensemble est généralement donné sous forme de contraintes logiques sur les marquages, c'est pourquoi on désigne également ξ comme l'ensemble des *contraintes terminales*. Nous pouvons ainsi définir le *langage terminal* du RdP qui est l'ensemble des séquences franchissables qui mènent à un marquage de l'ensemble ξ.

Dès lors que seuls les marquages terminaux sont acceptables, les marquages menant à un marquage de l'ensemble ξ doivent également faire partie de ξ. Il devient alors nécessaire de restreindre ce langage terminal de façon à ne prendre en compte que les marquages de l'ensemble ξ. Pour cela, on définit un sous-ensemble du langage terminal :

Définition 14 ***Centre du langage terminal***

Le ***centre du langage terminal*** $C_L(N, \xi)$ *d'un réseau de Petri étiqueté N est le sous-ensemble des mots du langage terminal définis à partir des marquages terminaux* ξ *par :*

$$C_L(N, M_0, \xi) = \{w = t_1 t_2 \ldots t_n \in T^* / si\ M_0(t_1 > M_1(t_2 > \ldots (t_n > M_n$$

$$alors\ M_0, M_1, \ldots M_n \in \xi\}$$

Il s'agit de contrôler à chaque étape que les contraintes spécifiées par ξ sont bien vérifiées. Le centre du langage se définit comme l'ensemble des séquences telles que les états intermédiaires sont tous dans ξ.

3.1.2 Extensions des Réseaux de Petri

Les réseaux Places/transitions que nous venons de présenter sont qualifiés d'autonomes. La caractéristique majeure des réseaux autonomes réside dans l'indéterminisme de leur évolution : nous pouvons décrire ce qui peut se passer mais nous ne pouvons pas savoir ce qui va effectivement se passer. Cela provient de la non prise en compte du contexte d'évolution du système. Par ailleurs, même si tous les systèmes parallèles sont relativement faciles à modéliser et malgré la puissance d'expressivité des RdP autonomes, certaines propriétés ne peuvent pas être exprimées. C'est par exemple le cas pour le test à zéro ou encore le fait de décider de la transition à franchir en cas de conflit. De même, dans le cadre de l'étude des systèmes temps réel, l'un des paradigmes fondamentaux du temps réel à savoir l'interaction temporelle avec le système, ne peut être pris en compte directement. C'est pourquoi il nous faut chercher parmi les nombreuses extensions des RdP celles permettant d'augmenter la puissance d'expressivité pour la prise en compte du temps comme facteur déterminant de l'évolution du système.

Enfin, lorsque les systèmes sont complexes, il est nécessaire d'essayer de réduire la taille de la modélisation qui devient trop conséquente pour pouvoir être exploitée dans les meilleures conditions.

3.1.2.1 Réseau de Petri colorés

Les réseaux de Petri colorés [Jen81] sont une abréviation des réseaux de Petri autonomes permettant d'en alléger la représentation graphique donc la lisibilité. Celle-ci est basée sur une coloration des jetons permettant de fusionner des parties similaires du réseau en une seule. Il existe plusieurs définitions des RdP colorés, de plus ou moins haut niveau [Jen94, Jen98] suivant le système que l'on souhaite modéliser. Nous ne nous intéressons qu'à l'expression la plus simple des RdP colorés.

Définition 15 ***Définition des Réseaux de Petri colorés***

Un réseau de Petri coloré *marqué est un 5-uplet,* $N =< P, T, C, W, M_0 >$

- *P,T sont définis de la même façon que pour les RdP autonomes,*
- *C est un ensemble fini de couleurs,*
- $W : (P \times T) \cup (T \times P) \times C \longrightarrow \mathbb{N}$ *est la fonction de valuation colorée, nous notons* $W(p_i, t_j, c_k)$ *la composante de couleur* $c_k \in C$ *du poids de l'arc* (p_i, t_j).
- $M : P \times C \longrightarrow \mathbb{N}$ *est le marquage initial coloré du réseau.* $M(p_i, c_j)$ *représente le nombre de jetons de couleur* c_j *du marquage de la place* p_i

---

L'apparition des couleurs sur les arcs et sur les jetons des places modifie la structure des marquages qui seront non plus des vecteurs mais des matrices avec une colonne par

couleur. Le franchissement des transitions reste analogue à celui des RdP autonomes classiques.

Définition 16 ***Franchissement des Réseaux de Petri colorés***

Soit M le marquage courant du réseau de Petri et $M(p_i, c_k)$ le marquage de la place p_i pour la couleur c_k alors :

- *Une transition $t_i \in T$ est* ***franchissable*** *à partir du marquage M si et seulement si $\forall p_j \in P, \forall c_k \in C,\ M(p_j, c_k) \geq W(p_j, t_i, c_k)$.*
- *Le franchissement de la transition t_i consomme pour chaque couleur c_k, $W(p_j, t_i, c_k)$ jetons et en produit $W(t_i, p_j, c_k)$ dans toute place p_j. Le marquage obtenu M' est défini par $M'(p_j, c_k) = M(p_j, c_k) - W(p_j, t_i, c_k) + W(t_i, p_j, c_k)$.*

La figure 3.2 montre un exemple de RdP coloré possédant 2 couleurs.

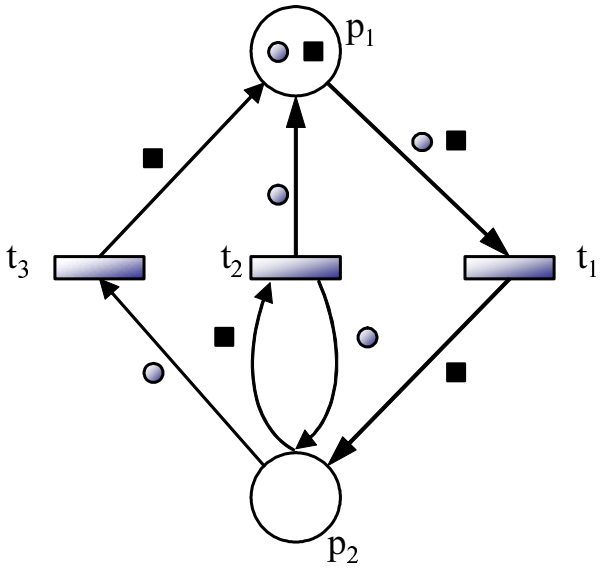

Fig. 3.2: Exemple d'un RdP coloré possédant 2 couleurs les ronds et les carrés. Le tir de la transition t_1 nécessite un rond et un carré dans la place p_1 et produit un carré dans la place p_2. La transition t_2 consomme un carré de la place p_2 et produit un rond en p_1 et en p_2. La transition t_3 nécessite un rond dans la place p_2 et produit un carré dans la place p_1. Si l'on franchit séquentiellement t_1 puis t_2 puis t_3 on se retrouve dans l'état initial.

Les RdP colorés sont autonomes puisque l'apport des couleurs ne change en rien l'indéterminisme des choix de tir des transitions franchissables. L'atout principal réside dans la simplification de représentation graphique des RDP usuels. Il est cependant toujours possible de transformer un RDP coloré en RdP classique (opération de dépliage), ce qui permet d'affirmer qu'ils possèdent la même puissance d'expression.

Par la suite nous présentons des extensions non autonomes des RdP. Pour ces réseaux, le critère de franchissabilité d'une transition dépend non seulement du marquage des

places mais également de critères supplémentaires. Ces derniers peuvent soit prendre en compte le contexte dans lequel le système évolue (Rdp synchronisé), soit dépendre d'une sémantique opérationnelle au niveau du modèle (règle de tir maximal, ensemble terminaux), soit enfin prendre en compte une dimension temporelle adjointe au réseau (RdP temporel et RdP temporisé).

3.1.2.2 Réseau de Petri synchronisé

Un réseau de Petri est dit synchronisé lorsque les tirs des transitions sont déclenchées par des événements (extérieurs). On prend donc en compte le contexte. Une transition est valide si les conditions spécifiées par les arcs sont vérifiées (il s'agit de la notion de franchissabilité des réseaux place-transitions) et ne peut être franchie que lorsque l'évènement déclencheur qui lui est associé se produit.

Définition 17 ***[MPS78]Définition des Réseaux de Petri synchronisé***

Nous appelons Réseau de Petri Synchronisé *le triplet* $S=< R,E,Sync >$ *où*

- *R est un Réseau de Petri autonome marqué,*
- *E est un ensemble d'évènements externes,*
- $Sync : T \longrightarrow E \cup \{e\}$ *où :*
  - *T est l'ensemble des transitions de R,*
  - *e est l'évènement toujours occurrent (évènement se produisant en permanence) .*

Nous parlons de RdP totalement synchronisé si aucune transition n'est associée à e.

Dans un RdP synchronisé, une transition valide n'est pas forcement franchissable. Elle devient franchissable quand l'évènement externe associé à la transition se produit : elle est alors immédiatement franchie. De cette façon, si plusieurs transitions sont simultanément valides et si elles sont toutes synchronisées sur des évènements distincts, le choix de celle qui sera franchie proviendra du contexte, puisque cela dépendra de l'identité du premier évènement qui se produira.

L'exemple de la figure 3.3 illustre la modélisation d'un atelier de coupe. Ici, nous associons le franchissement de la transition t_1 à l'évènement « arrivée d'une nouvelle commande ». Si la scie est disponible (présence d'un jeton dans la place « Scie disponible »), la transition « Début coupe » est automatiquement franchie puisque l'évènement toujours occurrent « e » lui est associé. La transition « Fin coupe » est finalement franchie uniquement lorsque l'opérateur ordonne l'arrêt de la scie ce qui se traduit par l'évènement « arrêt scie ».

La prise en compte des évènements extérieurs pour décider de la franchissabilité des transitions induit donc un fonctionnement événementiel. La notion de temps que nous recherchons est ici implicite puisque entièrement supportée par les dates d'occurrence des différents évènements. La mesurabilité temporelle ne peut être appliquée dans cette extension des RdP. En effet, la prise en compte des évènements n'est pas suffisante pour intégrer explicitement les contraintes temporelles du système de tâches.

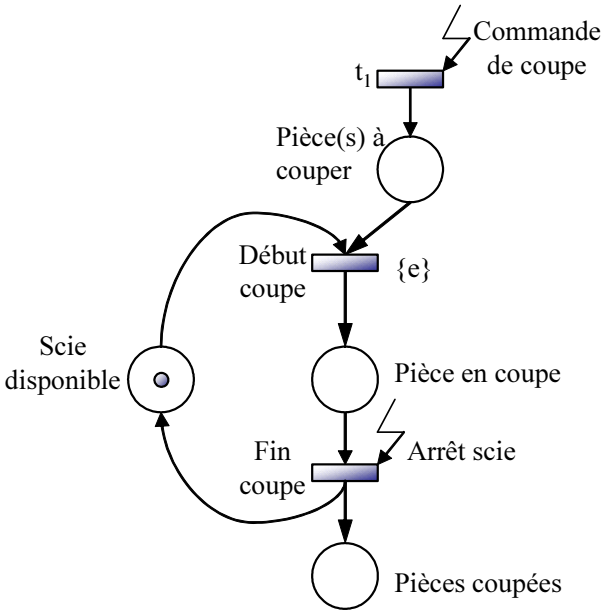

Fig. 3.3: Exemple d'un atelier de coupe modélisé par RdP synchronisé.

3.1.2.3 Réseau de Petri temporel

Les réseaux de Petri temporels (timed Petri nets) introduit par [MF76] permettent de localiser dans le temps le tir des transitions. Pour cela, ils associent un intervalle temporel de sensibilisation aux transitions. De plus, on suppose l'existence d'une horloge globale.

Définition 18 ***Définition des Réseaux de Petri temporels***

On appelle Réseau de Petri Temporel *le 5-uplet* $R =< P, T, T_{min}, T_{max}, W >$ *où*

- *P,T,W sont définis comme pour les RdP places-transitions*[a],
- $T_{min} : T \longrightarrow \mathbb{Q}^+$, $T_{min}(t_i)$ *est la date (statique) de tir au plus tôt de la transition* t_i
- $T_{max} : T \longrightarrow \mathbb{Q}^+ \cup \{\infty\}$, $T_{max}(t_i)$ *est la date (statique) de tir au plus tard de la transition* t_i
- $\forall t_i \in T, T_{max}(t_i) \geq T_{min}(t_i) \geq 0$,

On nomme $[T_{min}(t_i), T_{max}(t_i)]$ *l'intervalle de tir statique de la transition* t_i.

---

[a] $\mathbb{Q}^+$ représente l'ensemble des rationnels positif ou nuls

Les règles de franchissement des transitions imposent 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 des RdP classiques,
- le réseau doit avoir tiré la transition t_i avant que $T_{max}(t_i)$ unités de temps ne se

soient écoulées depuis sa sensibilisation, sauf bien sûr si une autre transition en conflit avec t_i l'a invalidée.

Les réseaux de Petri temporels ont été initialement proposés dans le but de modéliser des systèmes avec récupération d'erreurs. Leur analyse est rendue délicate par le fait qu'il existe un nombre de comportements infini étant donné que, pour un marquage donné, le nombre de couples possibles <Transition valide, Date de tir possible> est infini. Dans [BM82], une méthode d'étude exhaustive des comportements possibles des RdP temporels basée sur les classes de marquages est proposée. Les graphes de classes sont ainsi définis. Les sommets sont étiquetés par des couples <Marquage du réseau, Contraintes d'intervalles sur la sensibilisation des transitions franchissables>. Le graphe des classes permet d'avoir un aperçu de l'ensemble des états possibles du système.

La figure 3.4 représente la modélisation par RdP temporel d'une tâche périodique définie par une période de 10 secondes et une durée de traitement de 5 secondes. Les instances de cette tâche sont activée toutes les 10 secondes par la transition « Activer », puis si l'instance précédente est terminée (présence du jeton dans la place « terminé »), la nouvelle instance entre en exécution immédiatement par la transition « Début ». Une fois « en traitement » la fin de son exécution ne peut arriver avant les 5 secondes nécessaire à son exécution et ne peut également dépasser la période d'activation de 10 secondes pour ne pas engendrer de réentrance, la transition « Fin » est donc sensibilisé par l'intervalle $[5, 10]$.

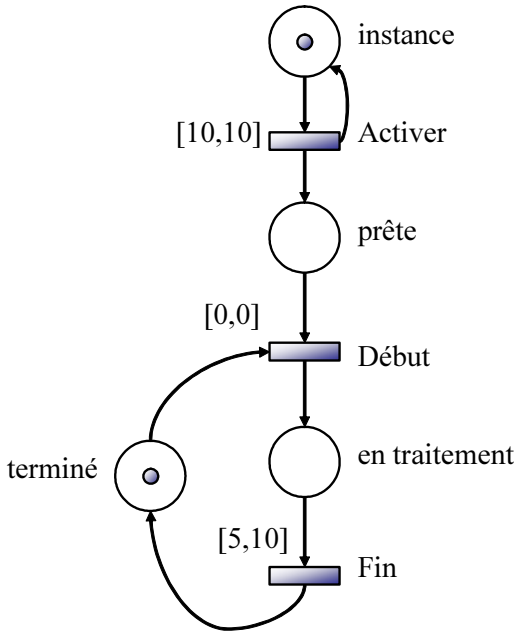

Fig. 3.4: Exemple d'un RdP temporel modélisant une tâche périodique de période 10 secondes et de durée d'exécution de 5 secondes.

Le principal avantage de ce modèle est sa puissance de représentation. En effet, les RdP temporels permettent de forcer un test à 0, chose non réalisable avec les RdP classiques. Les RdP temporels ont de ce fait la puissance de modélisation d'une machine de Turing. Cette extension des RdP s'avère être particulièrement bien adaptée pour l'étude de systèmes indéterministes temporisés, dans les cas par exemple où la connaissance de la durée des actions n'est que partielle. Cet indéterminisme se retrouve dans les bornes associées aux transitions. Il est à noter qu'il existe d'autres extensions basées sur les réseaux de Petri temporels mais dont la plupart repose sur des lois probabilistes [Jua99, Gal97], on les appelle les réseaux de Petri temporels stochastiques. La notion probabiliste ne correspond pas non plus au caractère déterministe dont nous avons besoin pour notre approche.

3.1.2.4 Réseau de Petri temporisé

Nous avons supposé que les paramètres temporels étaient connus et déterministes (à commencer par les durées). Pour prendre en compte ces durées, nous préférerons utiliser les *Réseaux de Petri temporisés* (Timed Petri Nets) qui ont été introduits par [Ram74, Chr83]. Le principe est d'associer une durée fixe aux transitions ou encore aux places du réseaux. On parle respectivement de réseaux T-temporisés ou P-temporisés (il est toutefois possible de mixer ces deux modèles).

Les définitions de ces réseaux sont intuitives puisqu'il s'agit d'adjoindre une durée soit aux transitions pour exprimer la durée de l'action, soit aux places pour exprimer la durée séparant deux actions. Il est important de noter qu'il est toujours possible de construire un RdP T-temporisé équivalent à un RdP P-temporisé.

Dans un RdP P-temporisé, quand une marque est déposée dans la place P_i, elle reste indisponible pendant le temps de sa temporisation. Une fois ce temps écoulé, elle devient disponible. Pour déterminer si une transition est valide, seules les marques disponibles sont prises en compte. Le marquage initial est supposé disponible au temps 0. Le franchissement d'une transition valide est supposé instantané.

Dans un RdP T-temporisé, 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 phase : les jetons nécessaires au franchissement de t_i sont retirés à une date δ indéterminée, puis, au bout de d_i unités de temps, les jetons produits par le tir de t_i sont déposés. La figure 3.5 illustre le fonctionnement des RdP T-temporisés.

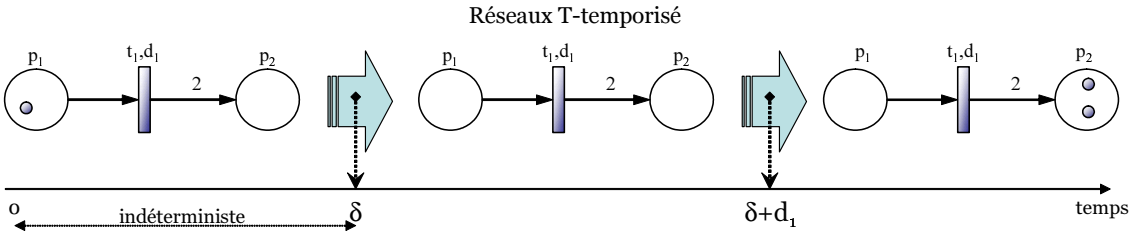

Fig. 3.5: Fonctionnement d'un RdP T-temporisé .

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 ou invalidée dès qu'elle est sensibilisée. Ainsi, les jetons sont consommés dès que la transition est sensibilisé et la production de jetons ne se produit qu'à la fin de la durée associée à la transition. Dans le cas des RdP P-temporisés, le fonctionnement à vitesse maximale s'exprime par une consommation immédiate après la validation de la transition. Lors que ces réseaux fonctionnent à vitesse maximale, la puissance d'expression est la même que celle d'une machine de Turing.

Cette approche semble en adéquation avec nos besoins. En effet, en utilisant la règle de tir maximal, nous enlevons tout indéterminisme sur le franchissement des transitions ce qui revient à modéliser les durées d'exécution déterministes des tâches temps réel. Toutefois, puisque nous associons ici une transition à une durée d'exécution, il n'est plus possible d'effectuer des préemptions de tâche. La représentation intuitive consistant à modéliser une tâche par une transition revient à modéliser uniquement des systèmes de tâches non préemptibles. Pour palier ce problème, nous pouvons raffiner cette modélisation en temporisant chaque transition par une durée unitaire. Une tâche est ainsi modélisée par une séquence de transitions associées à un quantum de temps unitaire. Il est ainsi possible de préempter les différentes tâches mais au détriment d'une surcharge d'informations à mémoriser. En effet, l'étude de ces réseaux reposent sur la construction d'un graphe des marquages temporisés qui nécessite de stocker en mémoire les marques disponibles, celles gelées et les durées résiduelles de tir des transitions enclenchées. Par conséquent, tous ces éléments alourdissent l'exploitation du RdP temporisé.

3.1.2.5 Réseau de Petri autonome avec la règle de tir maximal

Les RdP avec la règle de tir maximal ont la même structure que les RdP autonomes. Cependant, dans un RdP autonome, rien n'oblige une transition valide à être franchie, contrairement aux RdP avec la règle de tir maximal, qui intègrent un schéma opérationnel pour le franchissement des transitions. Ainsi, toute transition doit être franchie dès qu'elle est valide, à moins qu'une transition concurrente (en conflit) soit franchie à sa place. La règle de franchissement des transitions s'exprime alors de la façon suivante :

Définition 19 ***Franchissement des Réseaux de Petri autonomes avec la règle de tir maximal***

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 on franchit un ensemble* f *de transitions (franchissement simultané) tel que :*
– $f \subseteq F$
– $\forall t_i \in F \backslash f$, t_i *est en conflit avec au moins une transition de* f.

Sur la figure 3.6, 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 $f_2 = \{t_2\}$. Le fonctionnement en vitesse maximale obligeant f_1 ou f_2 a être franchi, les transitions

sont tirées tant que cela est possible à l'instar de l'ordonnancement conservatif des tâches où le processeur exécute les instances de tâches tant qu'il y en a dans la file des tâches actives.

La règle de tir maximal est comparable au fonctionnement à vitesse maximale pour les RdP temporisés. En effet, on peut noter qu'un RdP fonctionnant sous la règle de tir maximal est équivalent un RdP temporisé à vitesse maximale, dont toutes les transitions sont de durée unitaire [Sta90]. Pour toute modélisation de systèmes utilisant le temps, les RdP temporisés ou munis de la règle de tir maximal semblent être les plus indiqués dès lors que ces temps sont déterministes, ce qui est le cas dans les systèmes temps réel. Or, puisqu'ils possèdent la même expressivité, nous avons préféré utiliser les RdP à vitesse maximale.

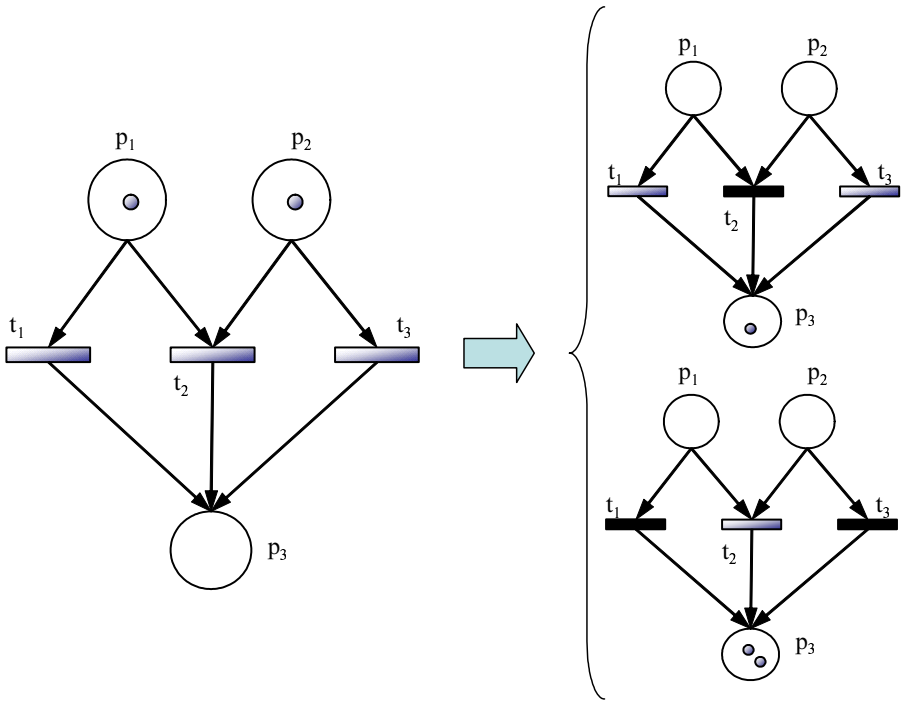

Fig. 3.6: Exemple d'un RdP autonome avec conflit fonctionnant avec la règle de tir maximal.

En effet, l'avantage de l'utilisation de ce modèle par rapport au modèle temporisé est sa simplicité de représentation et surtout d'implémentation. L'utilisation des RdP temporisés nécessite une prise en compte d'un vecteur des dates de fin des transitions dont le franchissement est en cours ce qui tend à alourdir son implémentation. Pour de plus ample informations sur ces RdP, on peut se référer aux travaux de [Sta90, JLKD86].

3.2 Modélisation des systèmes temps réel par Réseau de Petri

3.2.1 Présentation générale

Nous avons vu dans le chapitre 2 que dans le cas monoprocesseur, les techniques en-ligne d'ordonnancement d'applications temps réel multitâche étaient limitées notamment

en présence de ressources critiques. Dans le cas des systèmes temps réel fortement couplés (c'est à dire comportant de nombreux appels aux primitives temps réel), on a recours à des méthodes hors-ligne. Nous présentons ici une méthode hors-ligne d'ordonnancement basée sur une modélisation par Réseau de Petri initialement proposée dans [CGC96], puis étudiée par [GCG02]. Cette modélisation utilise une construction automatique d'un RdP à partir de la spécification d'un système de tâches périodiques temps réel.

Nous venons de voir que les réseaux de Petri avec la règle de tir maximal était les mieux adaptés pour cela puisqu'ils permettent à la fois de prendre en compte le temps et de modéliser de façon déterministe les durées liées au modèle temporel de tâches. Nous nous assurons également d'une puissance d'expression suffisante pour la modélisation du système et d'une relative simplicité de représentation et d'exploitation du graphe des marquages.

Cette modélisation par RdP se décompose en deux parties. La première définit une structure temporelle. Elle comporte une transition source, nommée *Horloge Globale* qui marque le temps (logique) en produisant des jetons dans les horloges locales des tâches et autant d'horloges locales que de tâches. Cette Horloge globale émet des tops horloge à intervalle régulier et nous considérerons que les tirs des transitions sont toujours synchronisés sur ces tops horloge. Les horloges locales permettent la réactivation périodique des tâches du système et le contrôle des contraintes temporelles qui leur sont imposées.

La deuxième partie de la modélisation consiste en une représentation du système de tâches qui définit finement le code décrivant les tâches, et notamment les contraintes structurelles imposées par les primitives temps réel comme les contraintes de précédence et d'exclusion. Nous utilisons une modélisation par boites aux lettres pour les communications et par sémaphores pour le partage de ressources. 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 fois, le système temps réel modélisé, nous construisons le graphe des marquages du RdP en tenant compte des contraintes temporelles. Et enfin nous procèdons à l'extraction d'un chemin du graphe des marquages symbolisant une séquence d'ordonnancement valide du système. La figure 3.7 présente le schéma de cette méthode.

Nous allons détailler par la suite les différentes étapes de cette approche en considérant l'étude d'une application temps réel composée de n tâches partageant r ressources et communicant avec m messages. Les ressources peuvent être utilisées en mode écriture et/ou en mode lecture et peuvent être multi-instances. De plus, nous considérons que cette application est à contraintes strictes et nous utilisons le modèle temporel issu de Liu Layland : soit S, le système des n tâches $\tau_1, \ldots, \tau_n$ composant l'application temps réel, nous avons alors $S = \{\langle r_i, C_i, D_i, T_i\rangle \, / pour \; 1 \leq i \leq n\}$

3.2.2 Structure temporelle

Pour mesurer le temps et l'adapter au RdP, nous adoptons un modèle discret du temps décrit dans [Kop92, Foh94]. Cette discrétisation du temps permet de définir une *Horloge Globale* qui est chargée d'égrainer le temps par la production de '*ticks*'(ou top horloge), une unité de temps correspondant à la granularité du système d'exploitation temps réel et plus précisément au quantum de temps minimum entre deux préemption successives.

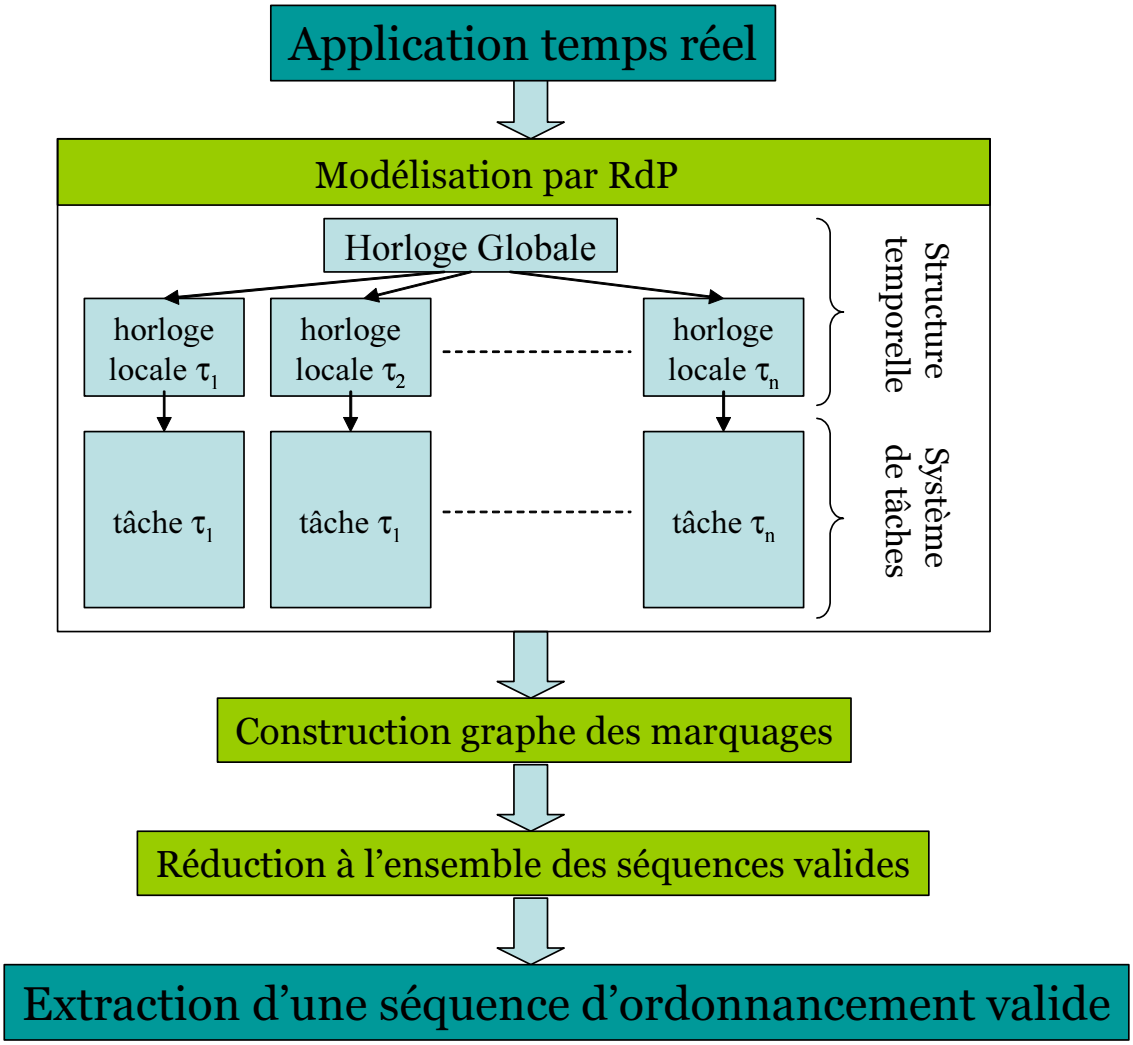

Fig. 3.7: Présentation générale de la méthode d'analyse hors ligne d'ordonnancement d'une application temps réel grâce à une modélisation par RdP.

Nous modélisons cette horloge globale par une transition source toujours franchissable noté *RTC* (pour *Real Time clock*). Celle-ci produit un jeton à chaque ticks. Il est ainsi possible de compter le temps et donc de modéliser les paramètres temporels liés à l'activation des tâches τ_i .

Pour modéliser la période T_i d'une tâche τ_i, il faut pouvoir comptabiliser T_i jetons pour activer la tâche τ_i. Pour cela, nous récupérons les jetons produits par RTC que nous stockons dans une place spécifique à la tâche τ_i que nous notons $Time_i$. $Time_i$ va donc comptabiliser le temps écoulé depuis la dernière activation. Lorsque $Time_i$ contient T_i jetons, alors une période de la tâche τ_i est respectée, il faut donc activer la tâche. Nous modélisons cette activation par une transition notée Clk_i, liée à $Time_i$ par un arc valué par T_i. Le couple $(Time_i, Clk_i)$ représente l'Horloge Locale de la tâche τ_i. Nous avons ainsi permis l'activation de τ_i, c'est à dire le tir de Clk_i, toutes les périodes T_i. Étant donnée la règle de tir maximal du réseau, les horloges locales obligent les transitions d'activations

Clk_i à tirer toutes les périodes T_i des tâches impliquant la production d'un jeton de couleur noté **a** (pour activation) toutes les T_i unités de temps dans la première place $Activ_i$ modélisant le corps de τ_i. La figure 3.8 illustre le fonctionnement de la structure temporelle en étendant le processus aux n tâches du système S.

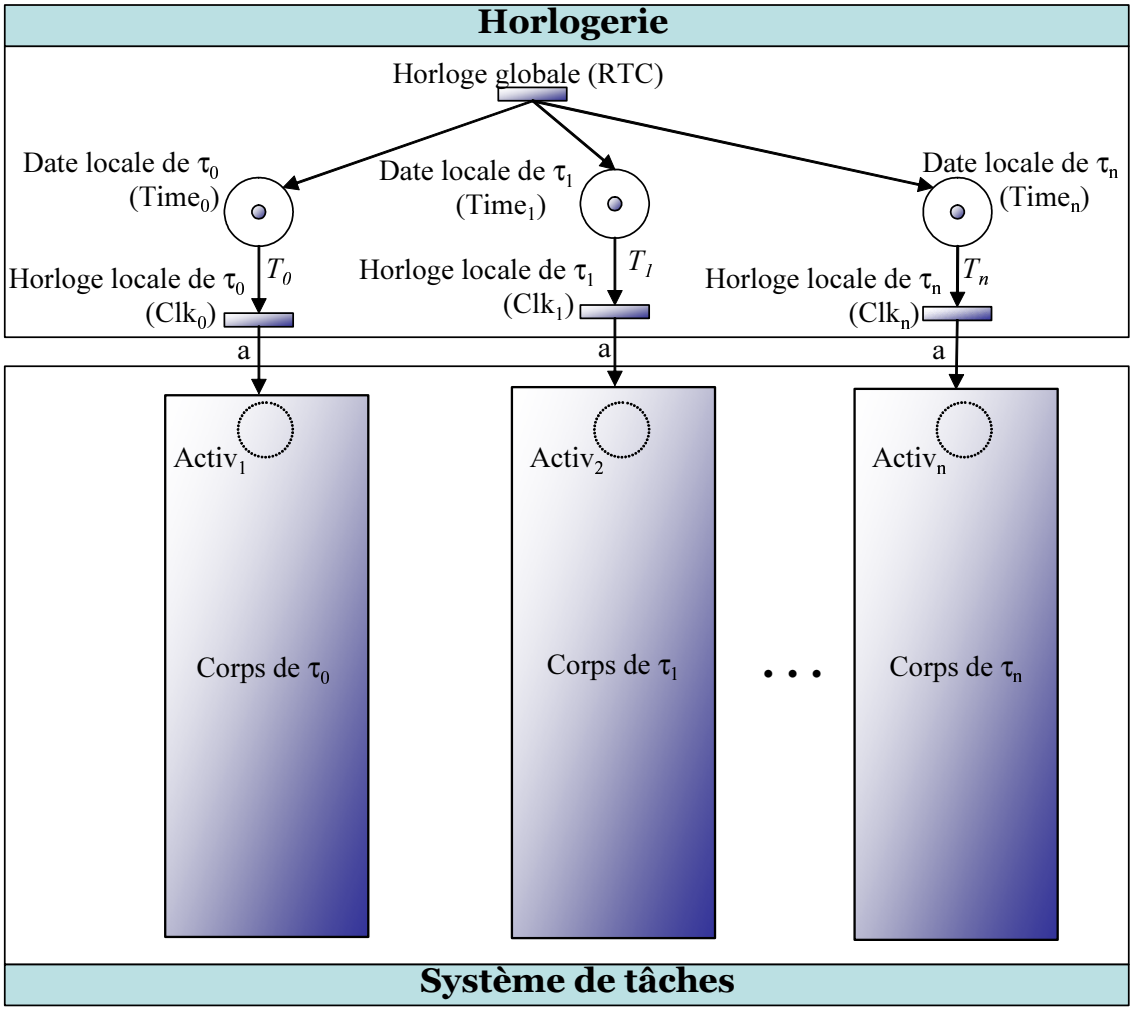

FIG. 3.8: Structure temporelle de la modélisation par RDP du système de tâche S.

À ce stade, cette structure temporelle ne permet que de définir des systèmes de tâches périodiques sans tenir compte de leur date de première activation. Pour ce faire, nous utilisons le marquage initial du RdP. Ainsi,

- si la date de réveil r_i de la tâche τ_i est telle que $r_i = 0$, alors τ_i doit être activée dès le premier tick de RTC. Or, l'activation d'une tâche telle que nous l'avons définie nécessite la production préalable d'un jeton coloré a à partir des horloges locales dans la place $Activ_i$ du corps des tâches. Nous choisissons donc de placer initialement dans $Activ_i$ un jeton coloré a.
- si $r_i \neq 0$ alors la production du premier jeton coloré a pour la tâche τ_i doit être retardée de r_i unités de temps (donc ici $a \notin M_0(Activ_i)$). Pour compter ce retard de

r_i unités de temps, il suffit de marquer initialement la place $Time_i$ par $M_0(Time_i) = T_i - r_i + 1$. Au bout de $r_i - 1$ ticks nous avons alors $M(Time_i) = T_i$ et la transition Clk_i peut ainsi être franchie : au temps r_i, nous avons bien un jeton a dans la place $Activ_i$. Toutefois, ce dernier cas ne fonctionne que si $r_i \leq T_i + 1$. Pour palier ce problème, il faut adjoindre à la place $Time_i$ dans le cas où $r_i > T_i + 1$, un mécanisme de décomptage éphémère qui ne servira qu'une seule fois au cours de la vie du réseau. Ce mécanisme est composé d'une place $Wait_i$ et d'une transition $Waiting_i$. Le but de cette transition est de consommer les jetons de $Time_i$ pour retarder le franchissement de Clk_i et donc l'activation de la tâche τ_i. Le nombre de jetons devant être consommés par $Waiting_i$ est ainsi égal à $r_i - T_i - 1$. C'est exactement le nombre de jetons que nous déposons dans la place $Wait_i$. La règle de tir maximal garantit que tant que $Wait_i$ est non vide, la transition $Waiting_i$ est tirée à chaque top horloge. Le marquage de la place $Time_i$ reste donc égale à 1. Ce n'est que lorsque $Wait_i$ est vide (au bout de $r_i - T_i - 1$ unités de temps) que la marquage de $Time_i$ commence à augmenter.

La transition $Waiting_i$ nécessite pour son franchissement un jeton de $Wait_i$ et un jeton de $Time_i$ et est en conflit structurel avec Clk_i. Ainsi, Clk_i ne peut être franchie que si $M(Wait_i) = 0$ puisqu'il faut que $Time_i$ engrange T_i jetons.

La figure 3.9 illustre la prise en compte des premières dates de réveils des tâches de S.

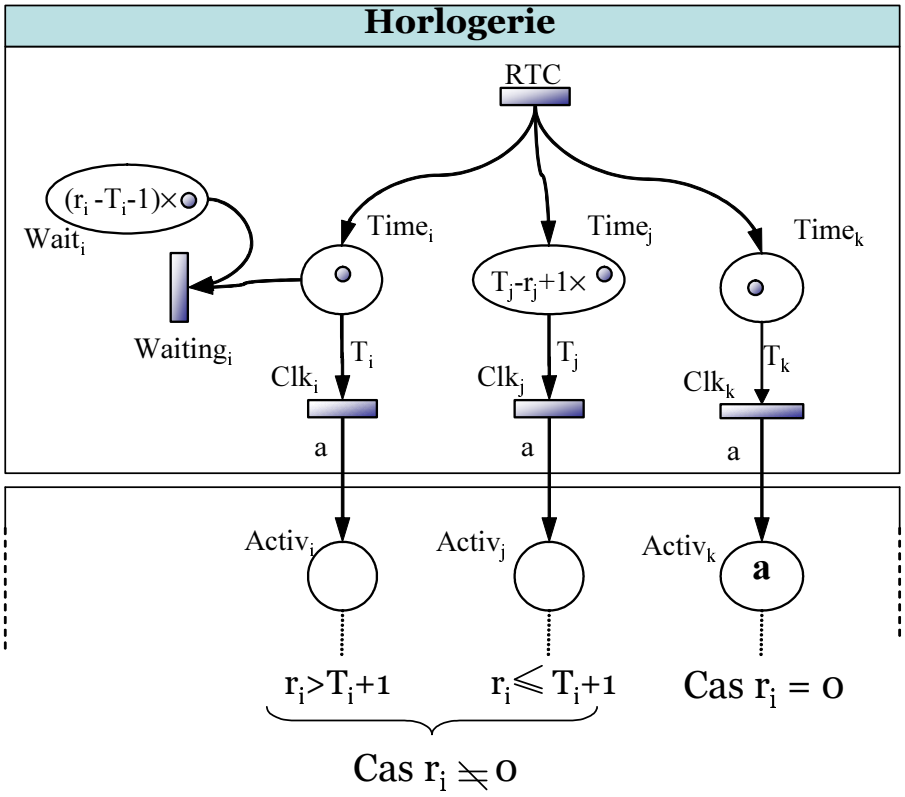

Fig. 3.9: Modélisation des dates de réveils r_i des tâches de S.

Nous avons vu comment activer les tâches en modélisant les deux paramètres temporels

r_i et T_i. Nous allons maintenant voir comment modéliser le corps des tâches ainsi que les paramètres C_i et D_i.

3.2.3 Système de tâches

La partie structurelle des tâches modélise le comportement d'exécution de chaque tâche. Nous avons vu que les RdP avec la règle de tir maximal sont équivalents aux réseaux temporisés avec des temporisations égales à 1. Nous considèrons donc qu'une unité de temps est nécessaire pour le franchissement d'une transition. Une tâche de durée d'exécution C_i se modélise alors par une série de C_i transitions.

Par ailleurs, nous souhaitons étudier, grâce à cette modélisation, des systèmes de tâches fortement couplés. Nous devons donc prendre en compte les primitives temps réel permettant le partage de ressources et la communications entre les tâches. Pour cela, puisque une durée d'exécution est modélisée par une série de transitions, il est possible de regrouper ces transition en les séparant par des instructions de bas niveau (du point de vue de l'accès au système) qui modéliseront les primitives temps réel liées à la gestion de ressources critiques et des communications. Ainsi, le paramètre temporel C_i peut être décrit plus finement en le découpant en :

- *blocs de durée* que nous notons $b_{i,j}$ pour le $j^{ième}$ bloc de durée de la tâche τ_i .
- primitives temps réel pour la gestion des ressources critiques. Celles-ci sont basées sur la notion de sémaphore et offrent les instructions Prendre_semaphore(R,n) et Rendre_semaphore(R,n) où R est la ressource et n le nombre d'instances de la ressource.
- primitives temps réel pour la gestion des communications en utilisant des boites aux lettres. Les primitives temps réel Deposer_BAL(bal) et Retirer_BAL(bal) où bal désigne une boite aux lettres, permettent de faire communiquer deux tâches par l'envoi d'un message dans la boîte bal.

Pour illustrer cela, considérons l'exemple d'une tâche définie par une durée d'exécution $C_i = 10$. Cette tâche nécessite également l'utilisation de la ressource R_j pendant 3 unités de temps à partir du temps 4. La figure 3.10 donne une description plus fine de l'exécution de la tâche τ_i en tenant compte des primitives temps réel de gestion de ressources.

```
Begin
bloc_i,1=4;
Prendre_Semaphore(R_j,1);
bloc_i,2=3;
Prendre_Semaphore(R_j,1);
bloc_i,3=3;
End;
```

FIG. 3.10: Exemple de la durée d'exécution d'une tâche de durée d'exécution de 10 unités de temps représenter par du pseudo-code intégrant des blocs de durée et des primitives temps réel.

3.2.3.1 Modélisation des blocs de durée

Les blocs sont indépendants et leur durée est calculée en nombre de ticks. Nous considérons que les instructions de boucle (en nombre borné d'itérations dans un contexte temps réel) sont déroulées et les instructions conditionnelles sont réunies [Bab96]. Ces blocs de durée sont modélisés par une série de transitions où chaque transition est reliée à la place processeur (arcs entrée et sortie).

Pour un système de tâches comportant des durées d'exécution relativement longues, une modélisation utilisant une série de C_i transitions pour chaque tâche τ_i deviendrait vite encombrante. C'est pourquoi nous simplifions cette modélisation en considérant les 2 cas suivant :

- si $C_i \leq 2$, alors on représente classiquement le corps de τ_i par le même nombre de transitions en série que la durée C_i.
- si $C_i > 2$, alors on modélise le corps de τ_i par 3 transitions : $T_{i,1}$, $T_{i,2}$, $T_{i,3}$. La première transition $T_{i,1}$ produit $C_i - 2$ jetons dans une place $P_{i,1}$. La transition $T_{i,2}$ consomme les jetons de $P_{i,1}$ un par un à chaque unité de temps et produit également un jeton à chaque franchissement dans une place $P_{i,2}$. Enfin, la troisième transition consomme les $C_i - 2$ jetons de $P_{i,2}$ en une unité de temps. Il y a donc $C_i - 2$ unités de temps consacrées au franchissement de $T_{i,2}$, 1 unité de temps pour $T_{i,1}$ et pour $T_{i,3}$. Nous modélisons bien la durée d'exécution C_i.

La figure 3.11 tirée de [Gro99] illustre la modélisation des durées d'exécution par RdP suivant ces 2 cas.

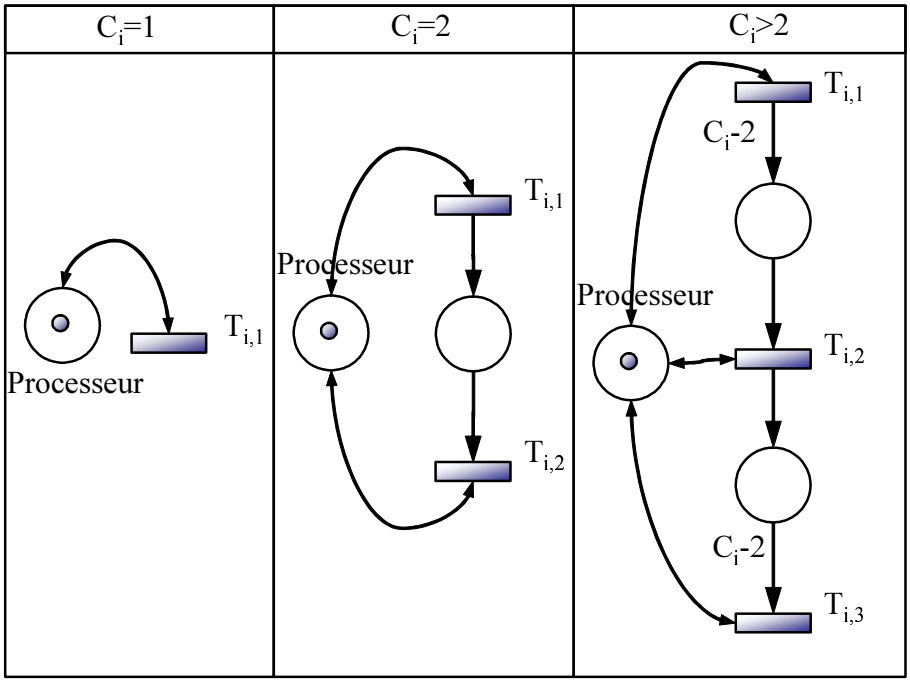

Fig. 3.11: Modélisation des durées d'exécution C_i des tâches de S et du processeur.

Nous utilisons ici la règle de tir maximal, ce qui revient à adopter une sémantique

du parallélisme maximal. Or, nous devons tenir compte de l'architecture cible qui est monoprocesseur. Cela signifie que nous devons imposer à la partie du réseau modélisant le système de tâches de fonctionner selon le principe de l'entrelacement. Pour cela, nous devons donc considérer le processeur comme une ressource nécessaire au tir de chacune des transitions formant la partie structurelle de S. Nous introduisons donc une place Processeur contenant un unique jeton (monoprocesseur). Cette place est reliée par deux arcs (un en entrée et un en sortie) à chaque transition du corps des tâches. De ce fait, toutes les transitions valides du système de tâches sont en conflit effectif, et une seule pourra être réellement tirée.

La figure 3.11 montre comment la place Processeur est reliée aux transitions constituant la tâche.

La préemption lors de l'exécution des tâches se fait naturellement dès lors que le processeur peut être consommé à chaque unité de temps par n'importe quelle transition du système de tâches. Toutefois, si la conception d'une tâche nécessite la non-préemption pendant tout ou partie de son exécution, il suffit de ne pas rendre le jeton processeur à la place Processeur entre la première transition de la partie non préemptible et la dernière : aucune tâche ne peut ainsi utiliser le jeton processeur (donc simuler une exécution) avant l'achèvement de cette exécution non préemptible.

Le franchissement de la dernière transition de chaque corps de tâche signale la fin de l'exécution de l'instance en cours. Pour prendre en compte cette terminaison, nous rajoutons un arc de la dernière transition à la place $Activ_i$ étiqueté par un jeton coloré noté **b**. Le rôle du jeton b est de représenter la fin d'une instance et de garantir la non réentrance, ainsi, b est déposé dans la place $Activ_i$ lorsque la dernière transition du corps de la tâche est tirée. Le tir de la première transition du corps de la tâche n'est donc possible que si la place $Activ_i$ contient :

- un jeton a : une nouvelle instance a été activée,
- un jeton b : l'instance précédente a terminé son exécution. Ce jeton est bien entendu placé dans chaque place $Activ_i$ du marquage initial M_0 pour pouvoir démarrer la première exécution.

La figure 3.12 représente la modélisation d'une application temps réel composée de deux tâches périodiques dont les paramètres temporels sont : $\tau_1 \langle 0, 2, 4, 4 \rangle$, $\tau_2 \langle 3, 1, 2, 2 \rangle$.

3.2.3.2 Modélisation des ressources critiques

Chaque ressource est modélisée par une place. Nous notons R_i la place représentant la $i^{ème}$ ressource du système qui contient initialement les m_i instances de la ressource. Si une tâche souhaite utiliser cette ressource, elle doit consommer les jetons de la place R_i pour simuler la prise du sémaphore protégeant cette ressource et restituer les jetons pris à la fin du traitement de la ressource, ce qui simule la restitution du sémaphore. Cette gestion des ressources permet de considérer deux cas d'utilisation :

- En mode Lecture : les arcs issus des transitions de la tâche τ_j, consommant et restituant les jetons de la ressource R_i, sont étiquetés par le nombre n_j d'instances nécessaires à son utilisation par la tâche concernée. Toutes les tâches voulant utiliser cette ressource peuvent le faire si le nombre d'instances restant est suffisant. Nous constatons alors qu'il peut y avoir k tâches utilisant la même ressource et se

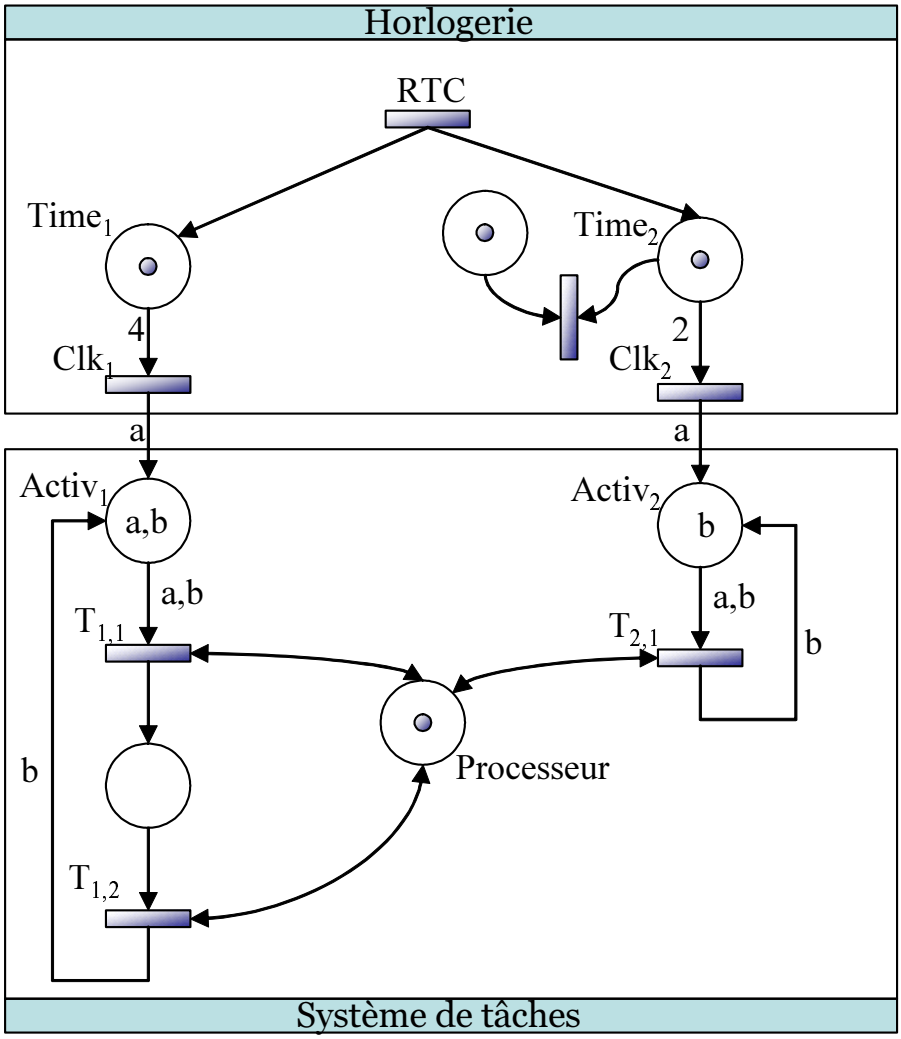

Fig. 3.12: Modélisation d'un exemple de système de tâches par réseau de Petri.

préemptant mutuellement si et seulement si $\sum_{j=1}^{k} n_j \leq m_i$.

– En mode Lecture/Ecriture : le mode d'accès en lecture reste identique au cas précédent en considérant tous les $n_j = 1$. Ainsi, chaque tâche voulant accéder en lecture ne peut consommer qu'un unique jeton. Le mode d'accès en Ecriture doit être effectué quant à lui uniquement lorsque aucune tâche n'accède en lecture à la ressource. Pour cela, il suffit d'étiqueter les arcs simulant la prise et la restitution de la ressource par m_i, c'est à dire le nombre maximal d'instances de la ressource. Dans ce cas aucun Lecteur ne pourra accéder à la ressource lorsqu'un Écrivain l'utilise.

La prise et la vente de sémaphores sont respectivement considérées comme commençant ou terminant un bloc. Elles sont donc fusionnées avec respectivement la première ou la dernière instruction du bloc. Nous les considérons ainsi de durées nulles. La figure 3.13 illustre la modélisation des ressources suivant ces deux modes d'utilisation.

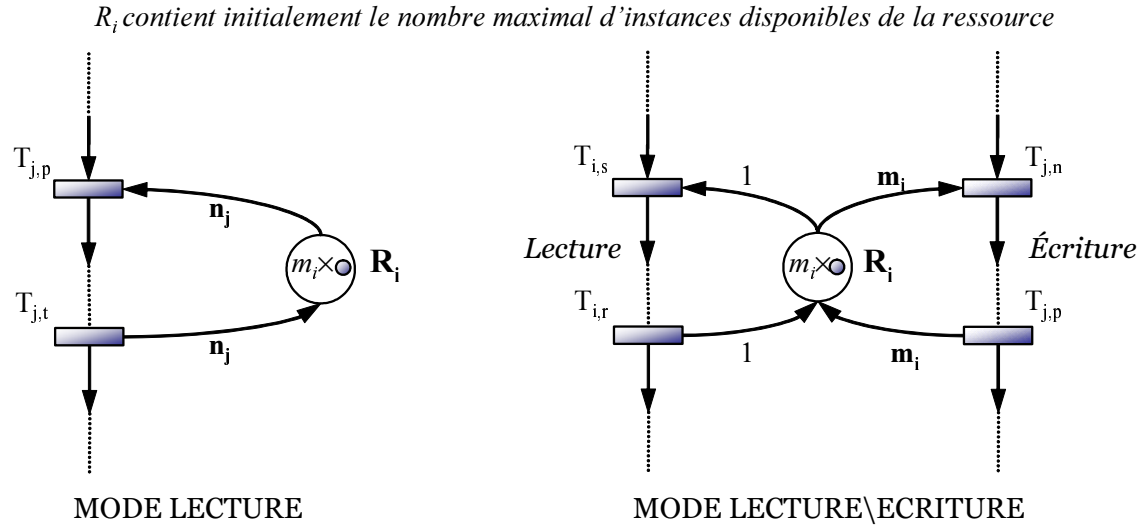

FIG. 3.13: Modélisation de l'utilisation des ressources en mode Lecture et Lecture/Ecriture.

3.2.3.3 Modélisation des communications

La modélisation des primitives temps réel pour la gestion des communications (supposées non bloquantes) utilise une place que nous notons BAL_i pour la $i^{ème}$ boite aux lettres. Dans le cas d'une boite aux lettres, le dépôt d'un jeton dans la place à partir d'une transition du corps d'une tâche τ_i représente l'envoi d'un message. La transition du corps de la tâche τ_j modélisant la réception du message doit alors consommer ce message. Nous considérons de plus que les envois et réceptions du message sont de durée nulle. Ce n'est généralement pas le cas dans la réalité, mais en fait, nous intégrons ces durées aux blocs de durée précédant ou suivant ces primitives d'envoi et de réception. La figure 3.14 illustre le fonctionnement la modélisation et le fonctionnement de l'envoi et la réception d'un message en utilisant une boite aux lettres.

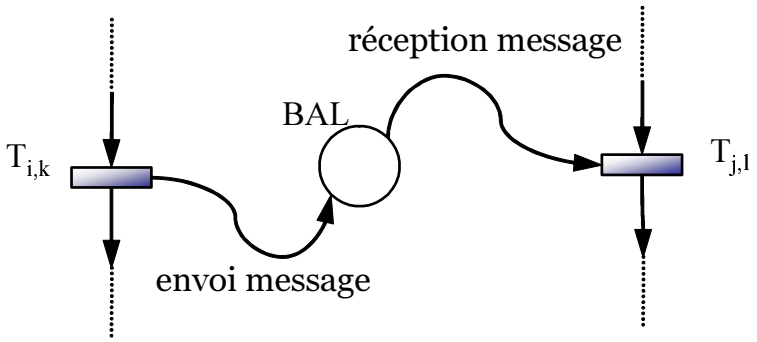

FIG. 3.14: Modélisation d'une communication entre deux tâches par boite aux lettres

3.2.4 Modélisation de l'inactivité du processeur

En l'état, le fonctionnement adopté permet de simuler uniquement les comportements conservatifs (au plus tôt) des applications temps réel. En effet, la règle du tir maximal du RdP force à tout instant le franchissement d'une des transitions valides formant le corps des tâches. Lorsqu'aucune des transitions du système de tâches n'est franchissable, un temps d'inactivité du processeur est simulé par le franchissement de la transition RTC, plus éventuellement de certaines transitions $Clock_i$, mais sans tir de transition du système de tâches. Or, comme nous l'avons vu dans le chapitre précédent, les ordonnancements conservatifs ne sont pas optimaux dès lors que des ressources critiques sont en jeu.

Pour permettre aux temps creux de survenir de façon arbitraire, nous modélisons les temps creux dits cycliques par l'ajout d'une *tâche oisive* notée τ_0. Celle-ci est définie par une période et un délai critique égaux à P (la métapériode de S), d'une charge de P(1-U) puisque par définition toutes les P unités de temps il y a $P \times U$ temps consacrés à l'exécution des tâches et donc $P - P \times U$ aux temps creux. L'ajout de cette tâche dans le système de tâche S permet d'obtenir une charge globale U' égale à 100%. Nous notons le système de tâche étendu $S' = S \cup < r_0, P(1-U), P, P >$. Cette nouvelle tâche est ici considérée comme une tâche du système S à part entière et est par conséquent modélisée de la même façon qu'une tâche du système de tâches. Le franchissement d'une transition de τ_0 correspondra alors à un temps creux du processeur et pourra survenir à tout moment.

De plus, dans le cas de tâche à départs différés, nous avons vu dans le chapitre précédent que des temps creux dits acyclique peuvent survenir. En appliquant la méthode de calcul du nombre de temps creux acyclique donnée par [CGG04]), il est possible d'obtenir le nombre ces temps que nous notons n_c. Pour en tenir compte dans la modélisation, on adjoint une place P_c contenant exactement n_c jetons, que nous relions à une transition T_c (étiquetée comme τ_0, pour ne pas différencier temps creux cycliques et acycliques dans le langage du RdP) symbolisant l'exécution de ces temps creux acycliques. Il faut alors monopoliser le processeur pour simuler leur prise en compte en le reliant à T_c de la même façon que les transitions d'une tâche du système S. La figure 3.15 représente la modélisation de la prise en compte des temps creux acycliques.

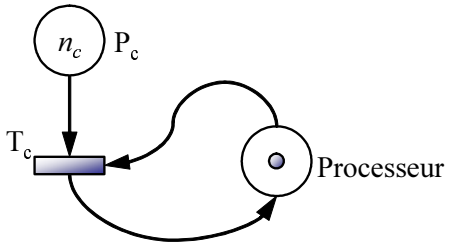

Fig. 3.15: Modélisation de la prise en compte des temps creux acycliques.

Nous notons que le paramètre r_0 de la tâche oisive τ_0 dépend de la présence de tâches à départs différés. En effet, s'il existe une tâche τ_i de S tel que $r_i \neq 0$ alors $r_0 = t_c + 1$ avec t_c la date du dernier temps creux acyclique [CGG04], sinon $r_0 = 0$.

L'ajout de τ_0 et de la gestion des temps creux acycliques permet à la modélisation de placer des temps creux du processeur à tous moments rendant l'obtention des séquences d'ordonnancement produites exhaustive.

3.2.5 Modélisation des contraintes temporelles

Nous venons de voir comment modéliser une application temps réel par RdP avec la règle de tir maximal. Nous avons vu comment tenir compte des paramètres temporels r_i, C_i, T_i des tâches. Il nous reste à prendre en compte le respect des délais critiques D_i. Pour cela, nous allons utiliser les jetons 'a' simulant l'activation des instances des tâches et les jetons 'b' simulera leur terminaison ainsi que les marquages des places $Time_i$.

Intuitivement, puisque nous ne voulons pas traiter les cas de ré-entrance (c'est à dire le cas où 2 instances successives d'une même tâche sont en exécution concurrente), il faut garantir que toute terminaison de tâches, c'est à dire la production du jeton b dans $Activ_i$ arrive avant la production de a (activation de la prochaine instance) dans cette même place $Activ_i$. Par conséquent, la marquage $M(Activ_i) = a$ correspond à un état non valide du réseau.

Lorsque la transition Clk_i est franchie, tous les jetons de la place $Time_i$ sont consommés (il reste donc 1 jeton produit au même moment par RTC). La place $Activ_i$ contient alors le jeton coloré b correspond soit au marquage initial, soit à la terminaison de l'instance précédente de la tâche et le jeton coloré a venant juste d'être produit par Clk_i. Par conséquent nous devons toujours avoir

$$M(Time_i) = 1 \Rightarrow M(Activ_i) = \{a, b\}$$

Dans le cas des tâches à départs différés, il n'y a pas dans le marquage initial de jeton a dans la place $Activ_i$, la tâche doit effectivement attendre d'être activée après r_i unités de temps. Par conséquent il existe également le cas où

$$M(Time_i) = 1 \Rightarrow M(Activ_i) = \{b\}$$

De plus, puisque chaque tâche τ_i possède une horloge locale grâce à la place $Time_i$ et la transition Clk_i, nous pouvons aisément vérifier le respect de son délai critique D_i. En effet, lorsque qu'il y a $D_i + 1$ jetons dans la place $Time_i$, cela signifie que depuis la dernière activation, D_i unités de temps se sont écoulées. Par conséquent, l'instance en cours doit avoir terminé son exécution ce qui se traduit par la présence du jeton b dans la place $Activ_i$. Ainsi, nous pouvons vérifier à tout moment que

$$M(Time_i) \geq D_i + 1 \Rightarrow M(Activ_i) = \{b\}$$

Puisque nous ne voulons nous intéresser qu'à des comportements valides d'ordonnancement des tâches, nous devons associer à la modélisation par RdP les contraintes terminales suivantes :

$$\boldsymbol{M(Time_i) = 1 \Rightarrow M(Activ_i) = \{a, b\} \text{ ou } \{b\}}$$

$$\boldsymbol{\wedge}$$

$$\boldsymbol{M(Time_i) \geq D_i + 1 \Rightarrow M(Activ_i) = \{b\}}$$

3.2.5.1 *Exemple d'une modélisation d'un système de tâche*

Afin d'illustrer cette méthode de modélisation par RdP muni de la règle de tir maximal, nous considérons le système de tâche S suivant :

$$S = \tau_1 < 0, 1, 4, 4 >, \tau_2 < 0, 2, 4, 4 >, \tau_3 < 0, 3, 16, 16 >$$

La figure 3.16 présente les corps de ces tâches. Les tâches τ_1 et τ_2 communiquent par boite aux lettres et les tâches τ_2 et τ_3 partagent une même ressource mono-instance R.

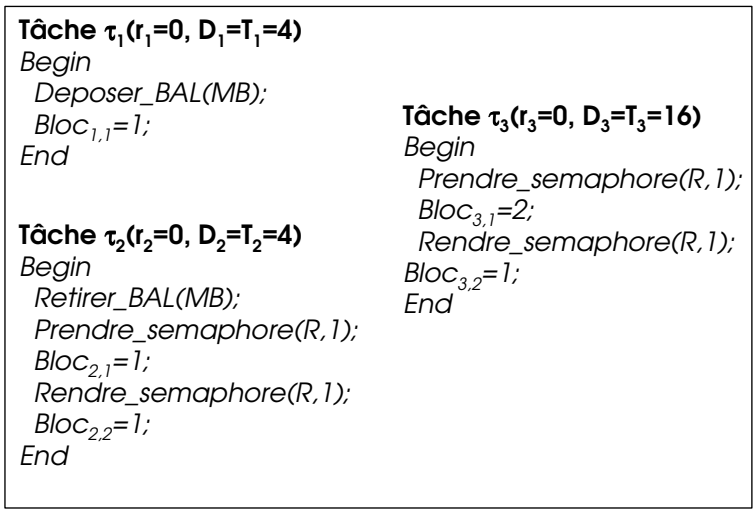

Fig. 3.16: Exemple d'un système de trois tâches.

La charge processeur étant de $U = \frac{1}{4} + \frac{2}{4} + \frac{3}{16} = \frac{15}{16} < 1$, nous devons rajouter une tâche oisive τ_0 pour simuler l'inactivité processeur. Comme nous l'avons vu, ses paramètres temporels sont basés sur la charge du système de tâche U. Nous définissons donc $r_0 = 0$ car les tâches de S sont à départs simultanés, D_i et C_i sont égaux à la métapériode de S c'est à dire $P = PPCM(4, 4, 16) = 16$, et enfin $C_i = P(1-U) = 16(1-\frac{15}{16}) = 1$. Nous ajoutons une tâche τ_0 à S pour obtenir le système de tâche étendu $S' = S \cup \{< 0, 1, 16, 16 >\}$. Les tâches étant à départs simultanés, il n'y a pas ici de temps creux acycliques.

Nous modélisons S' par le RdP avec la règle de tir maximal de la figure 3.17. Nous avons volontairement omis les arcs reliant le processeur aux transitions formant le corps des tâches pour ne pas alourdir la représentation.

3.3 Étude de la modélisation

Pour rappel, l'objectif de cette modélisation par RdP est de construire l'ensemble des séquences d'ordonnancement valides d'un système de tâches. Pour cela, nous associons à chaque transition une lettre de l'alphabet $\mathbb{A} = \{\tau_0, \ldots, \tau_n\}$ tel que la lettre τ_i soit affectée aux transitions modélisant la tâche τ_i. Toutes les autres transitions de la modélisation sont étiquetées par ϵ ce qui permet de les ignorer. Ainsi, un mot du langage du RdP est une séquence d'ordonnancement du système de tâches. Comme nous nous intéressons aux

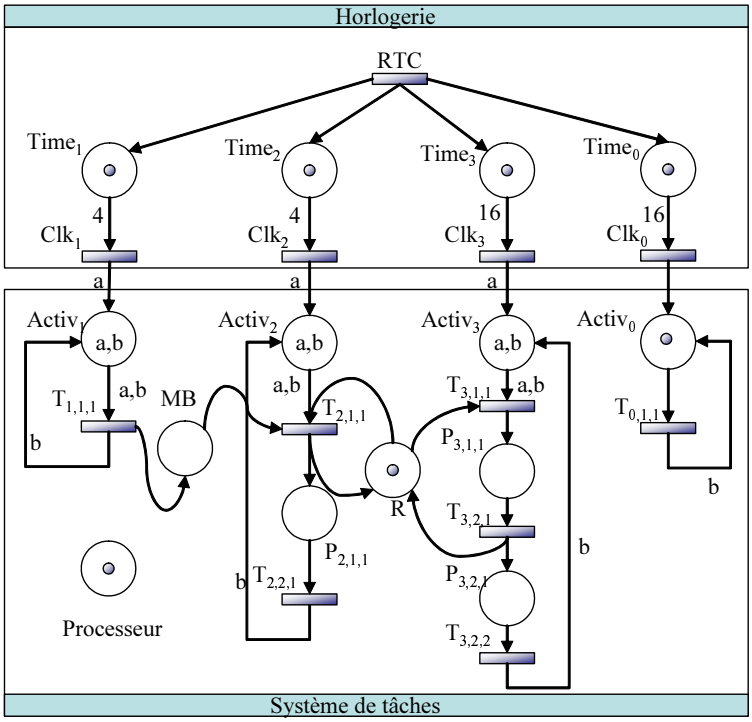

Fig. 3.17: Modélisation d'un système de trois tâches interagissantes à l'aide d'un RdP..

ordonnancements valides, nous devons choisir uniquement les mots pour lesquels chaque état respecte les contraintes terminales permettant de s'assurer de la validité temporelle. Ceci revient donc à définir le centre du langage du RdP modélisant le système de tâches (définition 14). De plus, puisqu' une séquence d'ordonnancement est par définition infinie, les mots du centre du langage doivent également être infinis. Nous cherchons donc à trouver l'ensemble des mots infinis du centre du langage. Pour cela, nous construisons le graphe des marquages du RdP réduits aux états composant les mots infinis. Ainsi, chaque mot, c'est à dire chaque séquence d'ordonnancement valide, est un chemin du graphe des marquages réduit du RdP modélisant le système de tâches. Par la suite, nous appellerons *Graphe d'Accessibilité*(noté *GA*), ce graphe des marquages réduit.

3.3.1 Graphe d'accessibilité

Comme toute étude exhaustive de problème NP-difficile induit une explosion combinatoire et comme le problème général de l'ordonnancement est NP-Difficile, nous nous devons d'analyser les limites aussi bien spatiales que combinatoires du graphe d'accessibilité. De prime abord, même si l'ensemble des mots infinis du RdP engendre obligatoirement une explosion combinatoire, il n'en va pas de même pour le graphe d'accessibilité. En effet, nous avons vu en 2.4 que les séquences d'ordonnancement sont périodiques et de période la metapériode P à partir de la date $t_c + 1$ (t_c étant la date du dernier temps creux acyclique). Nous en déduisons donc que les états du graphe d'accessibilité à la profondeur $t_c + 1 + k \times P$ sont les mêmes qu'à la profondeur $t_c + 1 + (k-1) \times P$ pour k>1. Par

conséquent, nous constatons que la construction du graphe d'accessibilité se limite à une profondeur égalé à $t_c + P + 1$ et nous pouvons ainsi nous ramener à une représentation du GA de taille bornée tout en tenant compte de tous les mots formant des séquences d'ordonnancement valides et infinies.

3.3.1.1 Propriétés du graphe d'accessibilité

Nous pouvons noter également que ce GA n'est pas une simple union de toutes les états successifs du RdP permettant d'exprimer l'ensembles des séquences d'ordonnancement valides. En effet, chaque état commun à plusieurs séquences d'ordonnancement n'est présent qu'une seul fois dans le graphe pour une même profondeur. Par conséquent, ce GA est moins effrayant du point de vue spatial qu'une technique d'énumération exhaustive de l'ensemble des séquences d'ordonnancement valides. Plus précisément, puisque nous devons construire un graphe défini par autant d'états qu'il y a d'étapes possibles durant l'exécution des tâches $\{\tau_0, \ldots, \tau_n\}$, nous pouvons nous ramener à représenter ce graphe dans un *hyperpavé* de dimension n+1, où chaque dimension dépend des caractéristiques temporelles des tâches.

Définition 20 ***Hyperpavé***

Un hyperpavé *de dimension d et de longueurs par dimension $L_1, \ldots, L_d$ est un graphe non orienté G=(V,E) défini par :*
- *si d=1, $V = \{v_0, \ldots, v_{L_1}\}$, et E est défini uniquement par les couples (v_{i-1}, v_i) pour $i = 1 \ldots L_1$,*
- *pour d>1, V est le produit cartésien de d hyperpavés de dimension 1.*

Le nombre de sommets d'un hyperpavé de dimension d et de longueurs $L_1, L_2, \ldots, L_d$ est donné par

$$\prod_{i=1}^{d} (L_i + 1)$$

Au vu de cette définition, un GA est totalement contenu dans un hyperpavé de dimension n+1 où les longueurs des dimensions $0, \ldots, n$ sont respectivement

$$\left\lceil \frac{(t_c + P + 1)\, C_i}{P_i} \right\rceil_{i=1 \ldots n}$$

et la longueur de la dernière dimension est $(C_0 + n_c)$ avec n_c le nombre de temps creux acycliques.

Nous remarquons toutefois que l'hyperpavé enveloppant la GA est considérablement surestimé comme nous le montre l'exemple de la figure 3.18 représentant la GA d'un système de tâches composé de 2 tâches à départs différés $\tau_1 \langle 0, 2, 6, 6 \rangle$ et $\tau_2 \langle 3, 2, 3, 3 \rangle$. Par simulation, on se rend compte qu'il existe 2 temps creux acycliques et la date t_c du dernier temps creux acyclique est 5. Par conséquent, le GA de ce système de 2 tâches est totalement contenu dans un hyperpavé de dimension 3 ayant pour longueur 4 pour la

tâche τ_1 et 8 pour la tâche τ_2. La tâche oisive quant à elle n'est présente uniquement que pour les 2 temps creux acyclique, par conséquent, la troisième dimension est de longueur 2.

Nous constatons alors que si l'hyperpavé englobe bien tous les états du GA, il n'en demeure pas moins, qu'il surestime considérablement la taille du GA. En effet, les échéances des tâches, l'exclusion mutuelle en présence de ressources ou encore les contraintes de précédences sont autant de facteur diminuant l'espace requis par le GA dans l'hyperpavé l'englobant. Toutefois, cette estimation de la taille du GA permet d'appréhender la complexité spatiale qu'engendre sa construction.

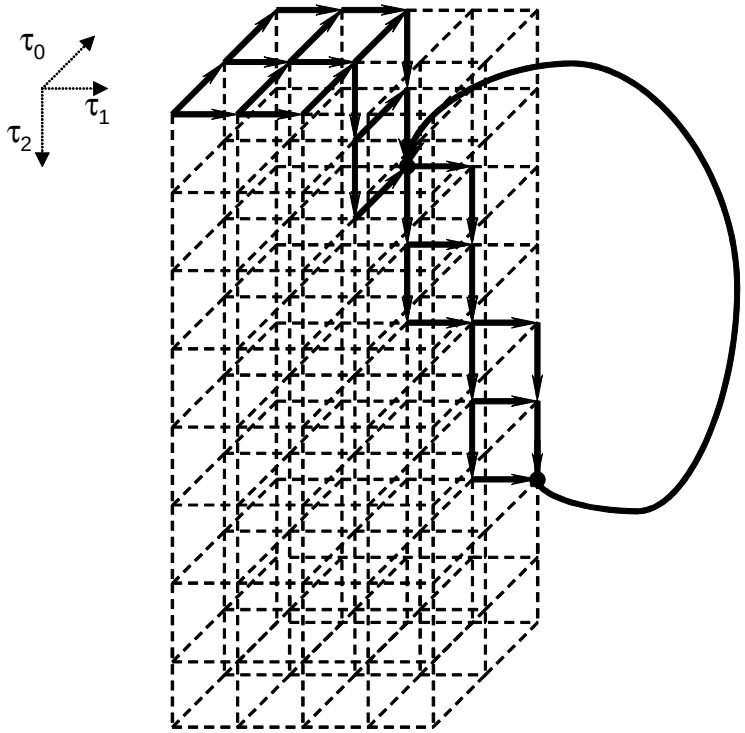

Fig. 3.18: Hyperpavé de dimension 3 englobant le graphe d'accessibilité résultant de la modélisation par RdP d'un système de tâche composé de 2 tâches.

3.3.1.2 Construction du graphe d'accessibilité

Afin d'obtenir les séquences d'ordonnancement valides d'un système de tâches modélisé par RdP, il nous faut préalablement extraire le centre du langage de RdP par la construction du GA. Ce dernier, comme nous venons de le voir, est de taille bornée par un hyperpavé dont les dimensions dépendent du système de tâches. La construction du GA repose sur la simulation de l'exécution du RdP et plus particulièrement son fonctionnement avec la règle de tir maximal. Or, à partir d'un état donnée du RdP, nous pouvons envisager deux approches de construction du GA :

– soit une construction en largeur d'abord où, partant de l'état initial, nous décrivons la liste des états successeurs, puis nous réitérons l'opération sur chacun de ces

successeurs *etc.*...

– soit une construction en profondeur où nous construisons d'abord complètement une séquence franchissable possible avant de passer à la suivante.

Construction en largeur

A partir du marquage initial M_0 formant le premier noeud du GA à la hauteur 0 du graphe (c'est à dire au temps 0 du RdP), nous déterminons tous les ensembles maximaux de transitions franchissables en accord avec la régle de tir maximal du RdP et produisant des marquages valides au sens des ensembles terminaux. Si k ensembles sont trouvés alors le premier noeud noté M_0 comportera k fils tous de hauteur 1. Pour chacun de ces k fils, nous réitérons l'opération jusqu'à la hauteur maximale du GA, c'est à dire $t_c + P + 1$. Un même marquage n'est évidemment construit qu'une seul fois. Lorsque cette profondeur maximale est atteinte, l'ensemble des séquences d'ordonnancement valides et par conséquent le centre du langage du RdP est totalement construit.

Construction en profondeur

Le premier noeud du GA, toujours de hauteur 0 comporte comme précédemment k ensemble maximaux de transitions franchissables $E_1, \ldots, E_k$. Nous choisissons alors E_1 et construisons le fils de M_0 de hauteur 1 noté $M_{1,1}$ tel que $M_0 \langle E_1 \rangle M_{1,1}$. L'opération est réitérée à partir de $M_{1,1}$. Nous construisons ainsi une succession de sommet à chaque étape chacun de hauteur strictement supérieure au précédent jusqu'à la profondeur maximale du GA. Lorsque celle ci est atteinte, nous obtenons une unique séquence d'ordonnancement valide. Pour construire la totalité du GA, il faut revenir au marquage de niveau inférieur et recommencer l'opération sans tenir compte des ensembles maximaux de transitions franchissables déjà exploités.

Le choix de la construction du GA dépend du type d'informations que nous souhaitons extraire du GA. Si seule l'existence d'une séquence d'ordonnancement est nécessaire, la construction en profondeur permet un gain considérable en temps puisqu'elle ne construit pas le GA dans sa totalité. Á l'opposé, si le choix de la séquence d'ordonnancement à extraire est basé sur des critères précis, l'ensemble du GA doit être construit et dans ce cas la construction en largeur est plus pertinente. En effet, une construction en profondeur du GA nécessite de mémoriser les sommets abandonnés lors des étapes précédentes pour être sûr de ne pas refaire les analyses déjà réalisées précédemment. Ceci correspond au cas où plusieurs chemins amènent le système dans le même état : le choix de la suppression ou de la conservation d'un sommet n'est effectué qu'en fonction du chemin en cours d'analyse. Avec une construction en largeur, la suppression d'un sommet du GA est définitive et n'est pas spécifique à un chemin particulier d'obtention du sommet. Cette suppression doit de plus être reportée sur son père et s'il ne possède plus de successeur il ne peut aboutir à un chemin valide et doit donc être supprimé également : c'est le principe du *retour arrière* (ou *backtracking*). Par conséquent, la construction complète du GA est préférable en utilisant la technique en largeur.

3.3.1.3 Optimisation de la construction du Graphe d'accessibilité

Comme nous l'avons vu, le GA souffre d'une taille relativement importante et sa construction nécessite, surtout dans le cas de construction en profondeur, de nombreux retours arrières. Il est alors intéressant d'intégrer des optimisations durant la phase de construction permettant de réduire la taille du GA tout en gardant la même puissance d'analyse d'ordonnançabilité.

Pour limiter les cas de retour arrière, nous avons recours à une heuristique basée sur la charge dynamique et la Latence dynamique. En effet, pour chaque instance d'une tâche τ_i , il est possible de savoir à l'avance si son échéance peut ou non être respectée par la suite. Pour cela, il faut vérifier si la laxité dynamique de τ_i à un instant t est supérieure ou égale à la charge totale dynamique résultant de la somme des charges dynamiques des tâches actives dont l'échéance est inférieure à celle de τ_i. Cette heuristique est vérifiée dans la phase de construction à chaque insertion d'un nouveau marquage. Nous nous assurons ainsi que le marquage venant d'être construit par l'exécution d'une tâche τ_i est valide puisque τ_i peut respecter son échéance pour l'instance en cours d'exécution. Cette technique engendre toutefois une construction d'une table de laxité dynamique devant être mis à jour pour chaque nouveau marquage inséré dans le GA.

Il est également possible de rajouter des heuristiques lors de la construction du GA pour permettre de réduire le nombre de marquages à insérer dans le GA. Une première heuristique appelé *contrainte de successeur*, permet de limiter les entrelacements inutiles entre deux tâches. En effet, deux tâches de charge 2 et 3 peuvent s'entrelacer 10 façons différentes, or l'entrelacement correspond du point de vue système à une préemption de tâche et par conséquent à un changement de contexte d'exécution pour passer de l'exécution d'une tâche à l'autre. Ces préemptions sont généralement coûteuses en temps processeur, c'est pourquoi il est d'ordinaire souhaitable de les minimiser dans une séquence d'ordonnancement. Pour éviter ces préemptions inutiles lors de la construction du GA, si le marquage actuel provient d'une exécution d'une tâche τ_i alors nous poursuivons l'exécution de la tâche τ_i excepté dans les cas suivant :

- une nouvelle tâche vient d'être (ré)activée,
- une tâche vient d'être débloquée par la libération d'une ressource ou en raison de l'arrivée d'un message,
- la tâche τ_i souhaite rentrer en section critique ou se met en attente d'un message,
- la tâche τ_i vient de terminer son exécution.

Seuls ces quatre cas peuvent engendrer une préemption. La figure 3.19 représente le gain apporté par cette heuristique sur un système de 2 tâches indépendantes à départs simultanés $\tau_1 \langle 0, 6, 12, 12\rangle$ et $\tau_2 \langle 0, 2, 4, 4\rangle$.

Une autre heuristique peut également être utilisée en précisant avant la construction les propriétés que doivent vérifier les séquences d'ordonnancement que nous souhaitons extraire du GA. Cette technique, basée sur l'ajout de *contraintes absolues a priori* permet de ne construire les marquages que s'ils vérifient les règles prédéfinies. Celles-ci peuvent porter sur la Laxité, le Temps de Réponse ou encore le Taux de Réaction (défini comme le rapport entre le Temps de Réponse et le délai critique pour une tâche donnée). Il est ainsi possible de construire le GA en n'autorisant que les noeuds dont le marquage associé respecte une condition comme un taux de réponse inférieur à une valeur arbitraire pour

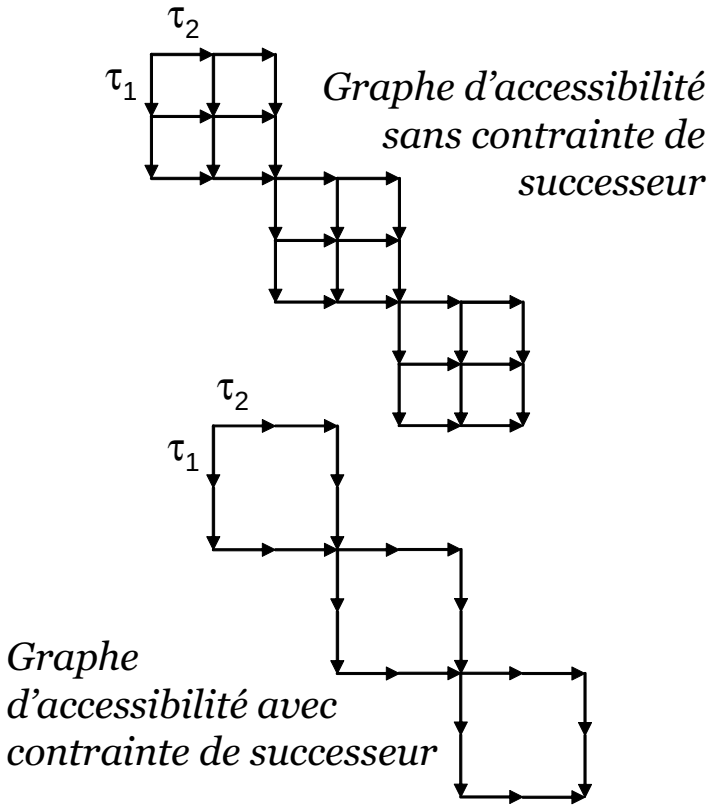

Fig. 3.19: Représentation du graphe d'accessibilité avec et sans contrainte de successeur pour un système de tâches composés de 2 tâches.

une ou plusieurs tâches du système de tâches modélisé. Ces contraintes sont vérifiées pour chaque noeud construit dans le GA. Si ces contraintes ne sont pas vérifiées, le noeud n'est pas construit. Ces nouvelles contraintes occasionnent toutefois de nombreux retours arrières qu'il convient une nouvelle fois de minimiser en s'appuyant sur la création comme précédemment d'une table de prévisions des critères définis par les contraintes a priori.

3.3.2 Extraction de séquences d'ordonnancement

Nous venons de voir comment construire la GA, c'est à dire l'ensemble des séquences d'ordonnancement valides en appliquant certaines optimisations globales pour réduire la taille du graphe. Nous voulons à présent extraire de ce graphe une ou plusieurs séquences d'ordonnancement. Pour cela, nous nous intéressons aux critères qualitatifs souhaités pour un sous ensemble de tâches de l'application temps réel. Le but de l'extraction de séquences consiste alors à choisir parmi les séquences d'ordonnancement du GA celles qui optimisent ces critères.

L'obtention des séquences optimales au vue de ces critères qualitatifs est linéaire en fonction de la taille du GA. Elle est basée sur un parcours de plus court chemin sur un graphe sans cycle avec une source et une destination, ce qui correspond au GA. Ce parcours n'est possible que si les arcs du GA ont été pondérés de façon à identifier les chemins de

l'état source (au temps 0) à l'état destination (de profondeur t_c+P+1) de coût minimum. [Gro99] a défini une méthode de parcours ascendant du GA permettant d'étiqueter les sommets en fonction de l'état des tâches suivant les critères qualitatifs prédéfinis. Ainsi, aucun parcours exhaustif et énumératif des séquences du GA n'est effectué : l'évaluation des sommets du GA à chaque hauteur, c'est à dire à chaque unité de temps, permet d'effectuer globalement l'évaluation des critères qualitatifs.

Soit $Opt = \tau_{i_1}, \ldots, \tau_{i_k}$ les tâches à optimiser, parmi les critères qualitatifs, nous trouvons :

Une minimisation du Temps de Réponse. Le but est de déterminer l'ensemble des séquences d'ordonnancement minimisant le temps de réponse maximal ou moyen pour le sous ensemble de tâches considéré. La pondération de chaque noeud s'effectue grâce à la formule suivante :

Soit

$$coût_{ResponseTime}(S) = \begin{cases} 0 & \text{si } \tau_j \notin Opt \text{ avec } S' \overset{\tau_j \in Opt}{\longmapsto} S \\ & \text{ou si } S' \overset{\tau_j \in Opt}{\longmapsto} S \text{ ne termine pas } \tau_j \\ sinon & \begin{cases} T_j \ si \ h - r_j \equiv O[T_j] \\ h - \left\lfloor \frac{h-r_j}{t_j} \right\rfloor T_j - r_j \ sinon \end{cases} \end{cases}$$

alors la pondération de chaque noeud est de :

$$coût(S) = min_{S' \in fils(S)} \left(coût(S') + coût_{ResponseTime}(S)\right)$$

Une maximisation de la Latence. Il s'agit de repérer les séquences d'ordonnancement maximisant la latence maximale ou moyenne pour un sous ensemble de tâches. La Latence d'une tâche se définit par son échéance moins son temps de réponse.

$$coût(S) = min_{S' \in fils(S)} \left(coût(S') - coût_{ResponseTime}(S) - d_j\right)$$
$$si \ S' \overset{\tau_j \in Opt}{\longmapsto} S$$

Une minimisation du taux de réaction. Le but est ici de trouver l'ensemble des séquences minimisant soit le maximum, soit la moyenne du rapport entre le temps de réponse et le délai critique pour chaque tâche du sous ensemble considéré. Le coût d'un sommet reprend ici la formule du temps de réponse mais en divisant le résultat par l'échéance critique pour les tâche de Opt :

$$coût(S) = min_{S' \in fils(S)} \left(coût(S') + \frac{coût_{ResponseTime}(S)}{d_j}\right)$$
$$si \ S' \overset{\tau_j \in Opt}{\longmapsto} S \ et \ t_j \in Opt$$

Une exécution au plus tôt. Il s'agit de chercher les séquences du GA permettant aux instances des tâches du sous ensemble d'être exécutées au plus tôt :

$$coût(S) = min_{S' \in fils(S)} \left(coût(S') + \begin{cases} \text{hauteur h de S si} & S' \overset{\tau \in Opt}{\longmapsto} S \\ 0 & sinon \end{cases}\right)$$

Il est également possible de considérer des critères qualitatifs non optimaux, comme la répartition des temps creux ou la minimisation de la gigue. En effet, ces derniers ont recours à des parcours multiples du GA et dépendent de paramètres supplémentaires, comme la date du premier temps creux rencontré pour le critère de répartition de temps creux.

3.4 Bibliographie

[Bab96] J.P. Babau. *Etude du comportement temporel des applications temps réel à contraintes strictes basées sur une analyse d'ordonnançabilité*. PhD thesis, thèse de doctorat d'informatique, LISI/ENSMA, 1996.

[BF86] E. Best and C. Fernandez. Notations and terminology on petri net theory. *Arbeitpapiere der Gesellschaft für Mathematik und Datenverarbeitung MBH 195*, page 29, Jan 1986.

[BM82] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time petri nets. *3rd European Workshop on Applications and Theory of Petri Nets*, Sept 1982.

[CG06] Annie Choquet-Geniet. *Les réseaux de Petri, un outil de modélisation*, volume ISBN=2100491474. Dunod, 2006.

[CGC96] A. Choquet-Geniet, D. Geniet, and F. Cottet. Exhaustive computation of the scheduled task execution sequences of a real-time application. In *Proc. of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 246–262, Uppsala, Sweden, 1996.

[CGG04] A. Choquet-Geniet and E. Grolleau. Minimal schedulability interval for real time systems of periodic tasks with offsets. In *Theoretical of Computer Sciences*, volume 310, pages 117–134, March 2004.

[CGVN93] A. Choquet-Geniet and G. Vidal-Naquet. *Réseaux de Petri et systèmes parallèles*. Ed. Armand Colin, 1993.

[Chr83] P. Chrétienne. *Les réseaux de Petri temporisés*. PhD thesis, Thèse dŠEtat, Université Paris VI, Juin 1983.

[Foh94] G. Fohler. *Flexibility in statically scheduled hard real-time sys-ems*. PhD thesis, University of Wien, April 1994.

[Gal97] L. Gallon. *Le modèle réseaux de Petri temporisés stochastiques : extensions et applications*. PhD thesis, Université Paul Sabatier, Dec 1997.

[GCG02] Emmanuel Grolleau and Annie Choquet-Geniet. Off-line computation of real-time schedules using petri nets. In *Discrete Event Dynamic Systems, DEDS*, volume 12(3), pages 311–333. Kluwer Academic Publishers, Jully 2002.

[Gro99] E. Grolleau. *Ordonnancement temps-réel hors-ligne optimal à l'aide de réseaux de Petri en environnement mono-processeur et multiprocesseur*. PhD thesis, ENSMA - Université de Poitiers, November 1999.

[HSSM68] A. Holt, H. Saint, R. Shapiro, and S. Marshall. Final report of the information system theory project. *Tech. Rep. RADC-TR-68-305, Rome Air Development Center, Griffis Air Force Base*, page 352, Sept 1968.

[Jen81] K. Jensen. Coloured petri nets and the invariant-method. *Theoretical Computer Science*, 14 :317–336, 1981.

[Jen94] K. Jensen. An introduction to the theoretical aspects of coloured petri nets. *A Decade of Concurrency, in Lecture Notes in Computer Science*, 803 :230–272, 1994.

[Jen96] K. Jensen. Coulored petri nets, basic concepts, analysis methos and practical use. *2nd edition, Monographs in theoretical Computer Science*, 1 :234, 1996.

[Jen98] K. Jensen. An introduction to the practical use of coloured petri nets. *Lectures on Petri Nets : Applications, in Lecture Notes in Computer Science*, 1492 :237–292, 1998.

[JLKD86] R. Janicki, P.E. Lauer, M. Koutny, and R. Devillers. Concurrent and maximally concurrent evolution of nonsequential systems. *Theoretical Computer Science*, 43(2-3) :213–238, 1986.

[Jua99] G. Juanole. Modélisation et évaluation du protocole mac du réseau can. *actes de ETR'99*, pages 187–200, 13-16 sept 1999.

[Kop92] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *12th Int. Conf. On Distributed Computing Systems*, pages 460–467, Japon, June 1992.

[MF76] P.M. Merlin and D.J. Farber. Recoverability of communication protocols - implications of a theoretical study. *IEEE Transactions on Communications*, pages 1036–1043, Sept 1976.

[MPS78] M. Moalla, J. Pulou, and J. Sifakis. Réseaux de petri synchronisés. *Journal RAIRO, Automatic Control Series*, vol. 12, n2 :103 :130, 1978.

[Mur89] T. Murata. Petri nets : properties, analysis and applications. *Invited Paper, Proc. Of the IEEE*, 17(4) :541–580, 1989.

[Pet62] C.A. Petri. Kommunikation mit automaten. *Bonn Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, English translation, 1966*, pages Vol.1, Suppl. 1, 1962.

[Pet80] 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 n4*, pages 21–27, 1980.

[Pet81] J.L. Peterson. Petri nets theory and the modeling of systems. *Prentice-Hall, Englewood Cliffs*, page 290, 1981.

[Ram74] C. Ramchandani. *Analysis of asynchronous concurrent systems by timed Petri nets*. PhD thesis, MIT, Cambridge, Feb 1974.

[Sta90] 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 :418–432, 1990.

Deuxième partie

Contribution

Chapitre Quatre

Prise en compte des variations de durées : Les Tâches Conditionnelles

Chapitre Quatre

Prise en compte des variations de durées : Les Tâches Conditionnelles

Les méthodes d'analyse d'ordonnancement reposent sur des modèles de tâches dont la durée d'exécution est considérée comme déterministe. Ce déterminisme est basé sur la prise en compte des pires durées d'exécution. Cette considération engendre malheureusement des anomalies d'ordonnancement et sur-contraint l'analyse d'ordonnancement, c'est pourquoi nous proposons un nouveau modèle temporel de tâches tenant compte des variations d'exécution des tâches et plus précisément des instructions conditionnelles, principale cause de ces fluctuations de durées. Nous définissons alors la notion d'arbre d'ordonnancement, implicitement dû à ce nouveau modèle, puis nous reformulons le problème de l'ordonnaçabilité en mettant en évidence l'ordonnançabilité Locale et Globale. En fin nous étudions les implications de ces concepts sur la durée de simulation optimale nécessaire à sa validation.

Sommaire

Chapitre Quatre

Prise en compte des variations de durées : Les Tâches Conditionnelles

Une grande majorité des outils de validation des applications Temps Réel reposent sur une étude des caractéristiques des tâches formant l'application, au travers d'un modèle de tâche. Ce dernier correspond le plus souvent à une abstraction déterministe du comportement des tâches s'appuyant sur les pires durées d'exécution. Ce modèle de tâche sert de base à l'analyse et la validation temporelle de l'application, comme nous l'avons vu au chapitre 2. Il s'agit, rappelons le, de :

- prouver l'existence d'une solution, c'est à dire d'une séquence d'ordonnancement respectant les contraintes temporelles. *C'est le problème de l'ordonnançabilité.*
- définir une politique d'ordonnancement. Cette politique pouvant être alors définie soit à l'aide d'un algorithme, soit par la donnée d'une séquence spécifique. *C'est le problème de l'ordonnancement.*
- prouver que la solution proposée est valide. *C'est le problème de la validation.*

Nous avons présenté au chapitre 2 les principaux algorithmes d'ordonnancement et les techniques de validation associés. Celles-ci utilisent la pire durée d'exécution. De plus, dans le cas où l'application manipule des primitives TR, le cumul de tous les retards possibles liés à l'utilisation de ces primitives doit être intégré aux pires durées d'exécution. Il s'ensuit que l'application analysée au travers du modèle de tâches est en fait beaucoup plus contrainte que ne l'est l'application effective, ce qui limite la portée de ces techniques (il y a un écart qui peut être significatif entre la modélisation et la réalité). Par ailleurs, nous avons vu également que des anomalies d'ordonnancement pouvaient survenir, lorsque que la durée d'exécution effective de certaines tâches est inférieure à la pire durée, ce qui interdit l'utilisation des techniques de validation par simulation. En ce qui concerne les stratégies hors ligne, les séquences obtenues sont elles aussi des contraintes et elles ne donnent pas une vue exacte de ce que sont en fait les différentes tâches. Un comportement fictif est décrit, correspondant à la plus longue durée d'exécution, couplée avec l'utilisation cumulée de toutes les primitives que manipule la tâche. Le plus souvent, la tâche n'aura jamais un tel comportement, en particulier, les fluctuations de durées d'exécution sont totalement masquées. Le prix du déterminisme est donc une augmentation des contraintes, et donc une perte de pertinence de l'analyse de l'application :

une application peut être qualifiée de non ordonnançable suivant les paramètres du modèle de tâche alors qu'elle l'est dans la réalité. Les analyses ont toutes pour socle le

modèle de tâche, lorsque celui-ci n'est plus représentatif de la réalité, mais n'en est qu'une approximation, il est clair que les solutions trouvées ne peuvent plus être performantes.

Pour palier ce problème, nous devons utiliser une abstraction la plus réaliste possible de l'application temps réel de façon à prendre en compte les fluctuations d'exécution d'une instance de tâche à une autre, tout en faisant en sorte de ne pas sur-contraindre la modélisation de l'application.

Nous pouvons trouver dans la littérature différentes méthodes permettant la prise en compte des fluctuations de durées mais le plus souvent ces méthodes sont utilisées pour répondre à des problèmes de tolérance aux fautes. Nous pouvons citer le *modèle à multi-représentations* [CHB79, CC91], qui associe à une tâche deux implémentations, l'une correspondant à une qualité de service optimale, mais de durée incertaine, et l'autre, de durée connue, mais correspondant à un service de moindre qualité ; le *modèle incrémental* [CL88], où une tâche est décomposée en deux parties, une partie essentielle qui doit absolument être exécutée, et une partie secondaire, permettant d'affiner les résultats produits, qui sera exécutée si le temps disponible est suffisant. Une autre démarche consiste à considérer une approche du pire temps de réponse comme dans [Ric02] ce qui permet de tenir compte de toutes les durées inférieures à la pire durée d'exécution. Toutefois cette méthode s'appuie sur un modèle sur-contraint pour ce qui concerne le partage de ressources.

Nous considérons ici seulement les fluctuations de durées liées à la présence d'instructions conditionnelles dans le code des tâches. En effet, ce type d'instruction est une cause majeure de fluctuation de durées. De plus cela permet d'étendre les possibilités des systèmes Temps Réel en attribuant plusieurs exécutions possibles pour chaque tâche du système. Cette pluralité sémantique du traitement de chaque tâche permet ainsi de concevoir des systèmes fonctionnant dans des modes différents suivant l'environnement dans lequel ils évoluent. Par exemple, pour faire face à une surcharge processeur en raison d'un grand nombre d'activations de tâches de type alarme, le passage dynamique dans un mode dégradé permet de soulager le processeur tout en s'assurant de la validité temporelle de l'ordonnancement des tâches. Nous pouvons également utiliser ce principe pour gérer plus finement la consommation d'énergie dans des systèmes embarqués ou encore assurer la coexistence d'une application Temps Réel et d'applications de second plan en jouant sur la durée de traitement des tâches de l'application Temps Réel pour offrir plus ou moins de temps processeurs aux applications d'arrière plan. Si les domaines d'application sont importants, il n'en demeure pas moins que dans ce contexte d'étude, l'analyse de faisabilité est NP-difficile [CET01]. Par ailleurs, le problème de l'ordonnancement est clairement défini dans le cas des durées uniques (il s'agit de produire des séquences possédant certaines propriétés). Mais il ne l'est plus si nous considérons de façon explicite les instructions conditionnelles, la notion de séquence d'ordonnancement n'étant plus adaptée. L'objectif de notre travail consiste à reformuler le problème de l'ordonnancement dans le cas où les tâches peuvent comporter des instructions conditionnelles. Il ne s'agit pas ici de définir une méthode permettant l'utilisation de tâches dites conditionnelles mais bien de spécifier les problèmes qu'engendre la prise en compte des instructions conditionnelles du point de vue de la modélisation des tâches, de leur ordonnancement et de leur analyse d'ordonnançabilité. Cette première étude permettra par la suite l'utilisation d'outils formels comme des automates ou réseaux de Petri pour modéliser ce type d'ap-

plications [PC04]. Dans un premier temps, nous devons compléter le modèle temporel de Liu-Layland afin de représenter explicitement les différents chemins d'exécution des tâches. Ensuite, nous transformons la notion de séquence d'ordonnancement en la notion d'arbre d'ordonnancement et nous caractérisons les arbres valides qui ont la propriété de garantir le respect des contraintes temporelles. Puis, nous reformulons le problème de l'ordonnançabilité : l'ordonnançabilité locale consiste à étudier de manière indépendante les différents chemins d'exécution, tandis que l'ordonnançabilité globale implique une vision globale de l'ensemble des comportements. Dans le cas de tâches indépendantes, nous montrons l'équivalence de ces deux notions alors qu'elles diffèrent dans le cas général. Afin de permettre l'utilisation de méthodes hors-ligne, il faut également définir la fenêtre temporelle dans laquelle il faut décrire le fonctionnement de l'application. En absence d'instructions conditionnelles, cette fenêtre correspond à la phase de montée en charge, suivie d'un cycle exécuté en régime permanent (cycle qui sera ensuite itéré). Nous montrons qu'en présence d'instructions conditionnelles et pour des tâches indépendantes, une telle fenêtre peut être définie et qu'il suffit de considérer la fenêtre obtenue en considérant simplement les alternatives dont les traitements sont les plus longs.

Dans une première partie, nous précisons nos hypothèses en posant clairement la problématique à partir d'un exemple. Nous définissons dans une deuxième partie, le modèle de tâches que nous utiliserons. Dans une troisième partie, nous introduisons la notion d'arbre d'ordonnancement. Nous définissons un opérateur de génération de ces arbres, puis nous précisons la notion de validité d'un arbre d'ordonnancement. La troisième partie étudie les principes de l'ordonnançabilité. Enfin, la dernière partie est consacrée à l'étude des durées de simulation et plus précisément à l'instant de début de cyclicité.

4.1 Problématique : exemple d'application non ordonnançable d'après un modèle de tâche classique

Considérons une application temps réel constituée de deux tâches τ_1 et τ_2 partageant chacune deux ressources R_1 et R_2. En usant du modèle classique issu de Liu et Layland, nous prenons les paramètres temporels suivants pour ces deux tâches : $\tau_1 \langle 0, 16, 32, 32\rangle$ et $\tau_2 \langle 0, 2, 4, 4\rangle$. De plus τ_1 utilise R_1 pendant 4 unités unités de temps (entre le temps 5 et 9 de sa durée d'exécution) et R_2 pendant 4 unités de temps (entre les temps 8 et 13 de sa durée d'exécution. La tâche τ_2, quant à elle, utilise également les deux ressources mais pendant une unité de temps chacune (R_1 puis R_2). La charge du processeur est ici de 100%. Suite aux 12 premières unités de temps, τ_2 s'exécute trois fois et τ_1 exécute 6 unités de temps. Donc à t=12, R_1 est allouée à τ_1 et τ_2 doit attendre que τ_1 libère la ressource R_1, ce qui peut se produire seulement au temps t=14. Mais, à ce moment là, la ressource R_2 sera allouée également à τ_1 donc τ_2 ne pourra s'exécuter que sur une unique unité de temps, puis devra attendre encore 3 unités de temps pour que la ressource R_2 soit libérée. τ_2 ratera ainsi son échéance.

Intéressons nous maintenant plus précisément à la durée d'exécution de la tâche τ_1. Pour cela nous regardons en détail le code de la tâche en pseudo langage ne faisant apparaître que les instructions conditionnelles de type IF THEN ELSE, les prises et libération de ressources. Le reste du code est représenté par un bloc de durée fixe. La

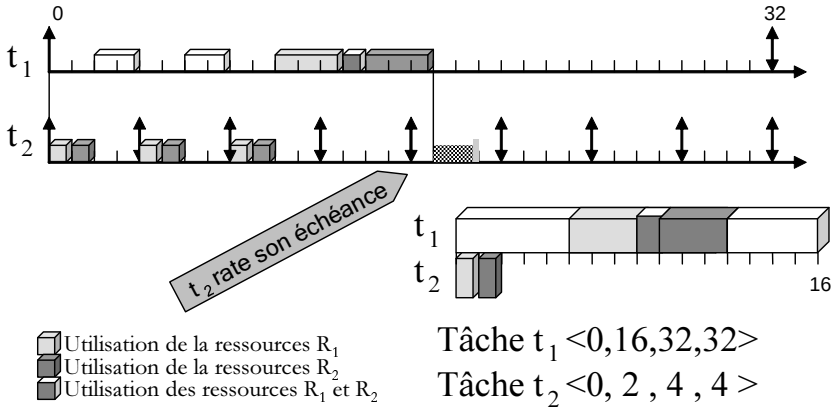

Fig. 4.1: Une application temps réel composée de 2 tâches $\tau_1 \langle 0, 16, 32, 32 \rangle$ et $\tau_0 \langle 0, 2, 4, 4 \rangle$ partageant 2 ressources non ordonnançable.

figure 4.2 illustre ce que pourrait être le code de la tâche τ_1. Á la lecture de ce code, nous pouvons immédiatement constater que les ressources R_1 et R_2 ne sont jamais utilisées en même temps. De plus la tâche 1 peut s'exécuter d'une instance à une autre de plusieurs façons totalement différentes. Enfin seules deux des cinq chemins d'exécutions utilisent une des ressources. Nous pouvons alors construire un arbre des exécutions possibles de la tâche 1 comme illustré sur la figure 4.2. Nous remarquons ainsi qu'il existe 5 comportements d'exécution liés à ce code et parmi ceux ci, la durée d'exécution varie de 5 unité de temps à 16 unités. Si nous utilisons le modèle des pires durées d'exécution, nous obtenons bien le modèle $\tau_1 \langle 0, 16, 32, 32 \rangle$ décrit figure 4.1, avec surmobilisation des ressources et il ne correspond en fait à aucune des exécutions réelles de la tâche. (Le passage de la tâche à l'abstraction déterministe est illustré figure 4.2).

Maintenant que nous avons montré que le modèle temporel de la tâche 1 correspond bien au code la tâche 1 illustré par la figure 4.2, nous pouvons tenter d'exploiter les différences constatés avec le pseudo code. Ainsi, la figure 4.3 montre que nous pouvons construire des séquences d'ordonnancement valides en cherchant à ordonnancer indépendamment chacun des 5 comportements d'exécution issus de l'arbre d'exécution de τ_1.

Au travers de cet exemple nous venons de mettre en évidence le fait qu'un modèle de tâche déterministe (pire cas) engendre :

- un modèle temporel de tâche surconstraint ce qui nuit à toutes les analyses d'ordonnançabilité,
- une utilisation non réaliste des ressources,
- une instabilité du système due aux anomalies d'ordonnancement liées aux variations de la durée d'exécution *cf* 2.3.4.

Par conséquent si nous ne tenons pas compte dès la phase de modélisation des tâches, des variations comportementales durant l'exécution des tâches, toutes les analyses d'ordonnançabilité, aussi performantes soit-elles, n'aboutiront qu'à un résultat non représentatif, voir erroné, vis à vis de la réalité.

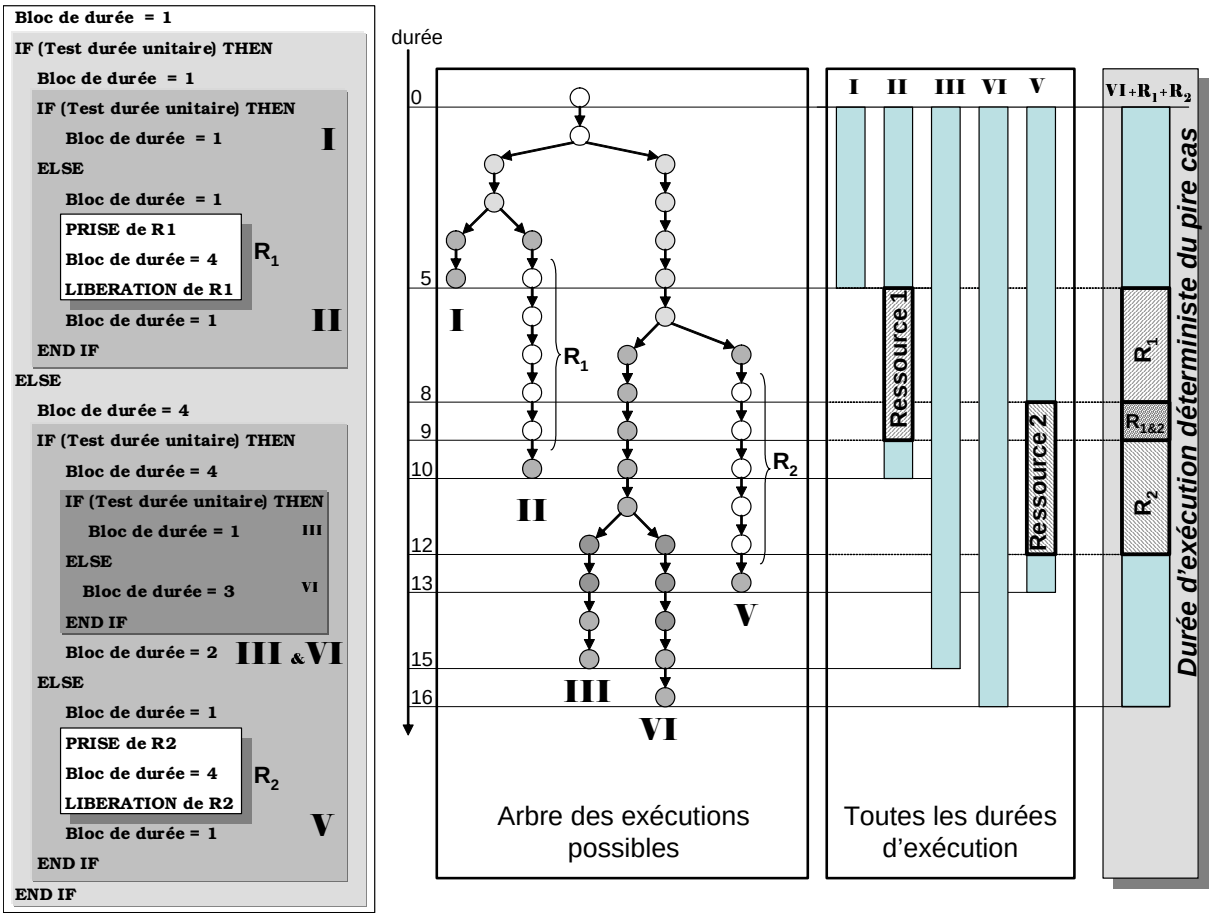

FIG. 4.2: Pire durée d'exécution d'une tâche τ_i en présence d'instructions conditionnelles. Le C_i résultant prend en compte le comportement le plus long, et mobilise les deux ressources.

4.2 Le modèle de tâche

Nous souhaitons prendre en compte les variations de durées d'exécution et d'utilisation de primitives TR induites par les instructions conditionnelles présentes dans le code des tâches. Nous devons par conséquent raffiner le modèle temporel classiquement utilisé par les outils d'analyse d'ordonnançabilité.

Nous considérons des applications constituées d'un ensemble de tâches $(\tau_1, \ldots, \tau_n)$ périodiques dont le code peut comporter des instructions conditionnelles et qui peuvent utiliser des primitives Temps Réel (prise en compte des notions de précédence, accès à des ressources critiques). Nous nous plaçons dans un environnement monoprocesseur. De manière classique, la tâche τ_i possède :

- une date de premier réveil r_i
- une période P_i
- un délai critique D_i

Les trois paramètres subsistent et demeurent inchangés. Par contre nous ne pouvons pas nous contenter du quatrième paramètre de la modélisation standard (C_i, la pire durée d'exécution). Le paramètre doit être raffiné pour mettre explicitement en évidence les

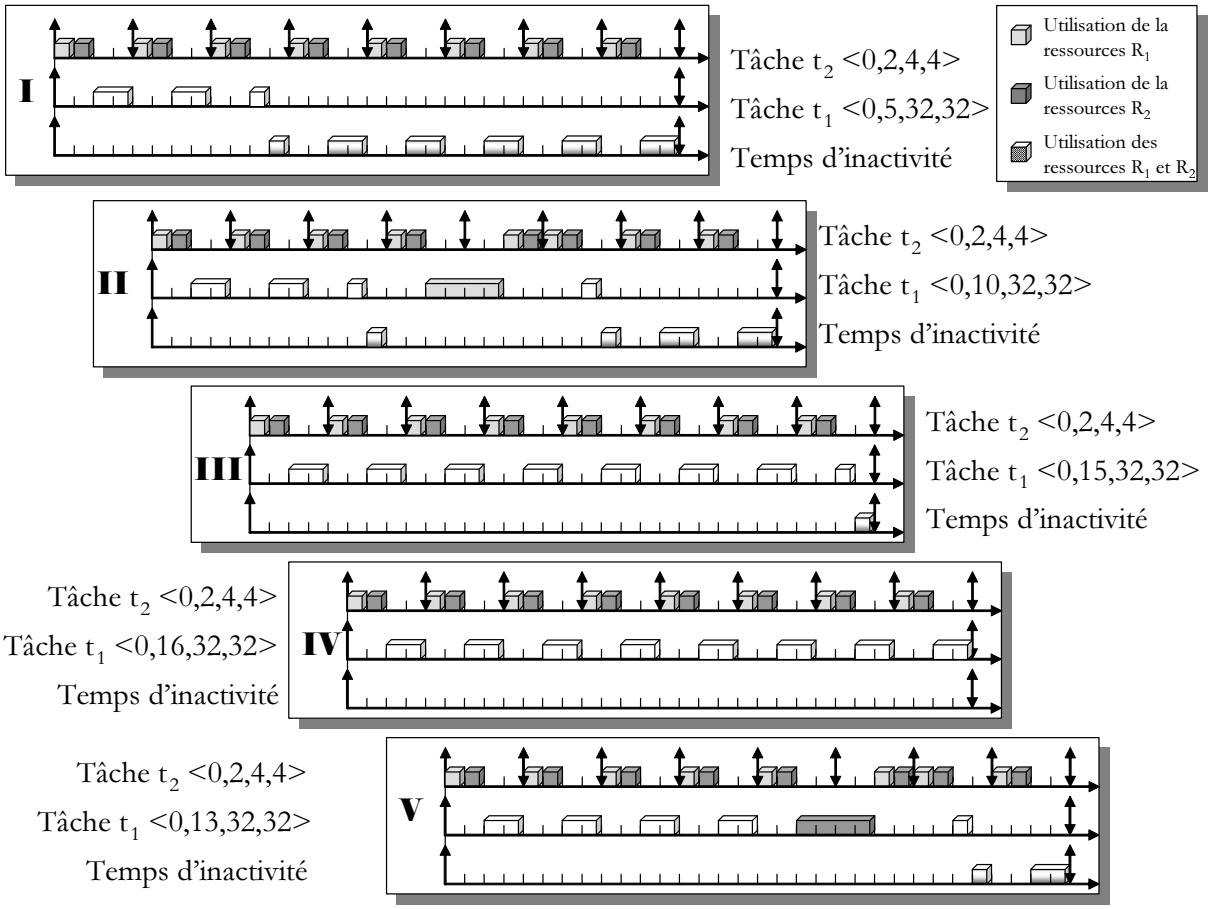

Fig. 4.3: Séquences d'ordonnancement valides suivant les 5 comportements d'exécution de la tâche 1 issus du pseudo code de la figure 4.2.

variations comportementales issues des instructions conditionnelles présentes dans le code des tâches.

4.2.1 Représentation des tâches

Dans un premier temps, nous devons représenter les différents comportements des tâches. Comme nous l'avons vu sur l'exemple précédent, nous pouvons construire un arbre des exécutions possibles d'une tâche directement à partir du pseudo code constitué des instructions BLOC-DE-DURÉE, IF-THEN-ELSE, Libération et Prise de Ressource. Pour cela, nous représentons chaque tâche τ_i par un arbre d'exécution de type flowchart valué sur l'alphabet $\alpha\beta_i$ tel que :

$$\alpha\beta_i = \{a_i, t_{i,j}\ avec\ 1 \leq j \leq k_i\}$$

où a_i représente une instruction élémentaire (d'une durée unitaire), $t_{i,j}$ représente la $j^{ème}$ instruction conditionnelle rencontrée dans le code de la tâche τ_i et k_i est le nombre de tests présents dans cette tâche. Les instructions conditionnelles $(t_{i,j})$ tiennent compte à

la fois du test positif (THEN) et du test négatif (ELSE). Nous définnissons en fait une représentation textuelle des arbres binaires que forment l'ensemble des comportements d'exécution des tâches. La tâche τ_1 de la figure 4.2 peut ainsi s'écrire

$$\tau_1 = a_1 t_{(1,1)} \left(a_1 t_{(1,2)} \left(a_1, a_1^6 \right), a_1^4 t_{(1,3)} \left(a_1^4 t_{(1,4)} \left(a_1^3, a_1^4 \right), a_1^6 \right) \right)$$

(par commodité d'écriture pour cet exemple, la notation $a_1 a_1 a_1 a_1 a_1 a_1$ est simplifiée par a_i^6).

Nous constatons que la durée d'exécution de chaque test conditionnel est considérée comme unitaire : si ce test dure plus d'une unité de temps, nous pouvons transférer sa durée sur le bloc de durée qui le suit.

Définition 21 ***Arbre d'exécution***

Soit une tâche τ_i comportant k_i instructions conditionnelles. Un arbre d'exécution définit l'ensemble des comportements d'exécution possibles de la tâche τ_i. Il est défini sur l'alphabet[a] *$\alpha\beta_i = \{a_i, t_{i,j}$ avec $1 \leq j \leq k_i\}$ et se construit de façon incrémentale comme suit :*

$$A_i = \tau_{i_0} t_{(i,1)} \left(\tau_{i_1}^+, \tau_{i_1}^- \right)$$

où τ_{i_0} est une séquence sans test, $\tau_{i_1}^+$ et $\tau_{i_1}^-$ sont deux arbres d'exécution correspondant aux exécutions de la tâche τ_i selon que le résultat du test $t_{(i,1)}$ est positif ou négatif.

---

[a] où a_i représente une instruction élémentaire et t_{i_j} représente le $j^{ème}$ test rencontré dans le code de la tâche

La figure 4.4 illustre sur un exemple simple cette notation arborescente.

4.2.2 Le modèle temporel

Nous définissons ensuite un modèle temporel dérivé de cette arborescence. La pire durée C_i est remplacée par un multi-ensemble formé par les durées d'exécution de chacune des branches de l'arborescence.

Définition 22 ***Modèle de tâche étendu***

Soit une tâche τ_i admettant $\wp_i$ comportements d'exécution, le modèle temporel étendu de τ_i est défini par $\langle r_i, \zeta_i, D_i, P_i \rangle$ où ζ_i est le multi-ensemble des durées des comportements de la tâche : $\zeta_i = \{ C_{i,j}$ avec $0 < j \leq \wp_i \}$.

L'exemple de la figure 4.4 illustre l'emploi des notations arborescentes et l'utilisation de ce nouveau modèle temporel. Nous pouvons constater que ce modèle étend celui de Liu-Layland. En effet, dans le cas où aucune instruction conditionnelle n'est présente dans le code des tâches, nous avons avec $\wp_i$=1 par conséquent, $\zeta_i = \{C_i\}$.

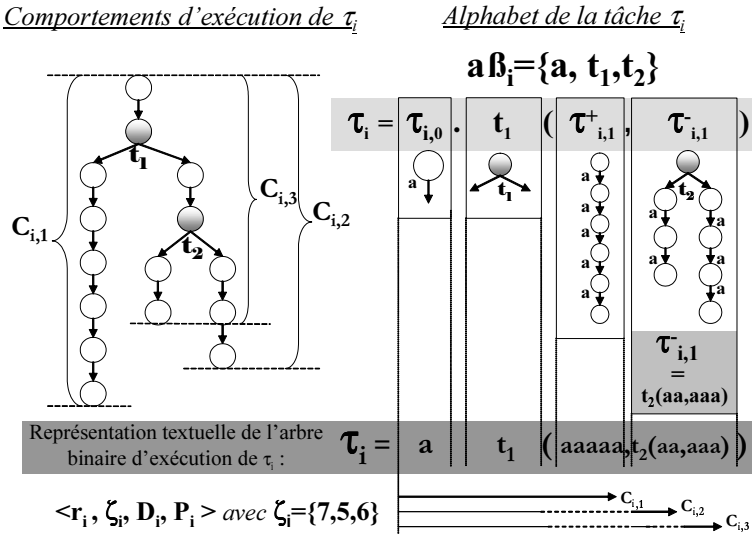

Fig. 4.4: La tâche τ_i possède un test conditionnel t_1 qui induit deux comportements d'exécution de durées 5 et 4 unités de temps. τ_{i_0} représente la partie commune à ces comportements et ne contient pas de test conditionnel. Les deux arbres $\tau^+_{i,1}$ et $\tau^-_{i,1}$ sont issus de l'instruction conditionnelle t_1 et représentent les deux comportements de la tâche τ_i suivant que le résultat du test est positif ou négatif.

4.3 Arbre d'ordonnancement

La définition du modèle étendu nécessite la redéfinition des méthodes d'analyse d'ordonnancement. En effet, la notion de séquence d'ordonnancement n'a plus de sens dès lors que le paramètre C_i peut varier dans le multi-ensemble ζ. Nous introduisons donc la notion d'arbre d'ordonnancement qui découle simplement de la prise en compte d'un arbre d'exécution.

Dans un premier temps, nous donnons une définition générale n'intégrant pas les paramètres temporels. Puis nous définissons un opérateur de construction systématique permettant d'intégrer les dates de premier réveil (r_i) et les périodes (T_i). La construction s'appuie sur les arbres d'exécution modélisant les tâches. Enfin, nous introduisons le critère de validité temporelle qui intègre les délais critiques (D_i).

4.3.1 Notion d'ordonnancement arborescent

Nous considérons par la suite que chaque tâche est définie par le modèle temporel étendu précédent. Nous nous intéressons à la notion d'ordonnancement d'un ensemble de n instances de tâche $(\tau_1, \ldots, \tau_n)$. Chaque tâche τ_i est associée à l'alphabet $\alpha\beta_i$ avec $\forall\, i, j \leq n$, $\alpha\beta_i \cap \alpha\beta_j = \emptyset$ pour $i \neq j$ et est caractérisée par son arborescence d'exécution. La représentation des tâches par des arbres d'exécution (et donc l'emploi du modèle temporel étendu) nous a tout naturellement conduit à introduire la notion d'ordonnancement arborescent. Nous redéfinissons l'alphabet $\alpha\beta_i$ de façon à mettre en évidence de façon explicite les composantes positives (THEN) et négatives (ELSE) des instructions conditionnelles. Ainsi, nous remplaçons chaque $t_{i,j}$ de $\alpha\beta_i$ par les lettres $t^+_{i,j}$ et $t^-_{i,j}$ correspondant respec-

tivement au résultat du test positif et négatif de l'instruction conditionnelle $t_{i,j}$. Nous définissons donc, de façon générale, les arbres d'ordonnancement par :

Définition 23

Un ***arbre d'ordonnancement*** $\mathcal{A}$ *de profondeur T d'un ensemble de n tâches* $(\tau_1, \ldots, \tau_n)$ *est un arbre dont les liens sont étiquetés par l'alphabet* $\alpha\beta_1 \cup \ldots \cup \alpha\beta_n \cup \{\Theta\}$ *où* Θ *représente une unité de temps d'inactivité du processeur tel que :*[a] [b]

Si η *est un noeud de* $\mathcal{A}$,

- $\|fils(\eta)\| \in \{0, 1, 2\}$.
- *Si* $\|fils(\eta)\| = 0$ *alors* η *est une feuille de* $\mathcal{A}$.
- *Si* $\|fils(\eta)\| = 1$ *alors* $\exists\ l \in \{a_i \text{ avec } 1 \leq i \leq n, \Theta\}$ *tel que* $\eta \xrightarrow{l} fils(\eta)$.
- *Si* $\|fils(\eta)\| = 2$ *alors* $\exists\ t^+_{i,j}$ *et* $t^-_{i,j}$ *avec* $1 \leq j \leq k_i\ ,\ 1 \leq i \leq n$ *tels que*

$$\left\{ l \text{ tel que } \eta \xrightarrow{l} fils(\eta) \right\} = \left\{ t^+_{i,j}, t^-_{i,j} \right\}$$

- *Toutes les feuilles de* $\mathcal{A}$ *ont même profondeur T.*

---

[a] Nous notons $fils(\eta)$ l'ensemble des fils de η dans $\mathcal{A}$.
[b] $\|\mathcal{E}\|$ =cardinal de $\mathcal{E}$

Lorsque le processeur exécute uniquement des instructions de type impératif (a_i), quelque soit les tâches exécutées, l'ordonnancement produit est une séquence. Dès lors qu'une instruction conditionnelle est exécutée, l'ordonnancement doit tenir compte des deux alternatives du test d'où la construction des deux branches de l'arbre d'ordonnancement correspondant aux tests positif et négatif. Nous permettons également au processeur de rester inactif. En particulier, lorsqu'il existe encore une branche de l'arbre d'ordonnancement qui doit exécuter une des tâches du système, les autre chemins de l'arbre qui n'ont plus de tâche à exécuter, sont complétés par Θ, indiquant que le processeur reste inactif. Ainsi, lorsque toutes les tâches sont terminées quelque soit les chemins d'exécution choisis, les feuilles de l'arbre d'ordonnancement sont toutes à la même profondeur.

4.3.2 Opérateurs de génération

Nous venons de spécifier la notion d'arbre d'ordonnancement, nous devons maintenant définir un protocole de génération de ces arbres qui doit tenir compte du comportement temporel d'activation des tâches. Nous présentons ainsi un opérateur de génération des arbres d'ordonnancement à partir des arbres d'exécution des tâches, en intégrant les paramètres r_i, ζ_i et P_i.

Nous définissons cet opérateur de manière incrémentale. Notons que cet opérateur ne construit que des arbres d'ordonnancement de type conservatif (le processeur ne sera inactif que s'il n'y a plus d'instance de tâche prête à être exécutée).

Rappelons que nous considérons une application constituée de n tâches périodiques $(\tau_1, \ldots, \tau_n)$, (τ_i étant caractérisée par $\langle r_i, \zeta_i, D_i, P_i \rangle$ et par la donnée d'une arborescence).

Nous appelons T la taille de la fenêtre temporelle sur laquelle nous voulons observer les comportements de l'application. Il s'en suit que T est la profondeur des arbres générés. Pour commencer nous introduisons quelques notations :

- $(\tau_{1,n_1}^{d_1}, \tau_{2,n_2}^{d_2}, \ldots, \tau_{m,n_m}^{d_m})$ est la liste des instances de tâches dont le réveil se situe avant T
- d_i est la date d'activation de $\tau_{i,n_i}^{d_i}$ (d_i est de la forme $r_{n_i} + k \cdot \ P_{n_i}$)
- n_i est le numéro de la tâche concernée.
- $e_i = \left\lceil \frac{T-r_i}{P_i} \right\rceil$ le nombre d'instances de τ_i activées dans la fenêtre $[0, T]$.
- $m = \sum_i^n e_i$ est le nombre total d'instances activées avant T, toutes tâches confondues.

et nous supposons que $d_1 \leq d_2 \leq \ldots \leq d_m$.

Nous définissons deux opérateurs : le premier génère l'ensemble des arbres d'ordonnancement à partir d'un ensemble d'instances des tâches et le second s'applique à un ensemble de tâches périodiques.

Définition 24 ***Opérateurs de génération***

L'opérateur

$\otimes_T$: {*Ensemble des instances des tâches réveillées avant* T} $\longmapsto$ {*Ensemble des arbres d'ordonnancement de profondeur* T}

est défini par :

- $\otimes_0((\tau_{i,n_i}^{d_i})_{1\leq i\leq m}) = \emptyset$
- $\otimes_{t>0}(\emptyset) = \{\Theta.\lambda,\ /\lambda \in \otimes_{t-1}(\emptyset)\}$
- *Si* $d_1 > 0$ *alors* $\otimes_T(\tau_{i,n_i}^{d_i}) = \left\{\Theta.\lambda,\ /\lambda \in \otimes_{T-1}(\tau_{i,n_i}^{d_i-1})\right\}$
  Nous introduisons un temps d'inactivité processeur puisqu'aucune instance de tâche n'est réveillée.
- *Si* $\exists\ j > 0$ *tels que :* $d_1 = d_2 = \ldots = d_j = 0,\ d_{j+1} > 0$ *alors :*
  - *Si* $\tau_{n_i} = \left(a_{n_i}.\tau'_{n_i}\right)\ \forall i = 1 \ldots j$ *(toutes les instances commencent par une instruction déclarative) alors :*

$$\otimes_T(\tau_{i,n_i}^{d_i}) = \bigcup_{l=1}^{j}\left\{a_{n_l}.\lambda,\ /\lambda \in \otimes_{T-1}\left(\tau'^{0}_{l,n_l}, \left(\tau^0_{1,n_1}, \ldots, \tau^0_{j,n_j}\right) \setminus \left(\tau^0_{l,n_l}\right), \tau^{d_{j+1}-1}_{j+1,n_{j+1}}, \ldots, \tau^{d_m-1}_{m,n_m}\right)\right\}$$

  .
  - *Sinon, quitte à réorganiser la liste, nous pouvons supposer que :* $\exists\ 1 \leq u \leq j$ *tel que :*

$$\begin{cases} \tau_{n_l} = \left(t_{n_l}(\tau^+_{n_l}, \tau^-_{n_l})\right) & (1 \leq l \leq u) \\ \tau_{n_l} = \left(a_{n_l}.\tau'_{n_l}\right) & (u < l \leq j) \end{cases}$$

  alors :

$$\begin{aligned} \otimes_T(\tau_{i,n_i}^{d_i}) = \ & \textstyle\bigcup_{l=1}^{u}\Big\{\left(t^+_{n_l}.\lambda^+, t^-_{n_l}.\lambda^-\right) \text{ tel que} \\ & \lambda^+ \in \otimes_{T-1}\left(\tau^{+^0}_{l,n_l}, \left(\tau^0_{1,n_1}, \ldots, \tau^0_{u,n_u}\right) \setminus \left(\tau^0_{l,n_l}\right),\ \tau^{d_{j+1}-1}_{j+1,n_{j+1}}, \ldots, \tau^{d_m-1}_{m,n_m}\right), \\ & \lambda^- \in \otimes_{T-1}\left(\tau^{-^0}_{l,n_l}, \left(\tau^0_{1,n_1}, \ldots, \tau^0_{u,n_u}\right) \setminus \left(\tau^0_{l,n_l}\right),\ \tau^{d_{j+1}-1}_{j+1,n_{j+1}}, \ldots, \tau^{d_m-1}_{m,n_m}\right)\Big\} \\ \cup\ & \textstyle\bigcup_{l=u+1}^{j}\left\{a_{n_l}.\lambda, \text{ tel que } \lambda \in \otimes_{T-1}\left(\tau'^{0}_{l,n_l}, \left(\tau^0_{1,n_1}, \ldots, \tau^0_{j,n_j}\right) \setminus \left(\tau^0_{l,n_l}\right), \tau^{d_{j+1}-1}_{j+1,n_{j+1}}, \ldots, \tau^{d_m-1}_{m,n_m}\right)\right\} \end{aligned}$$

Nous définissons ensuite l'opérateur

$\widetilde{\otimes}_T$: {*Ensemble des applications Temps Réel multi-tâches*} $\longmapsto$ {*Ensemble des arbres d'ordonnancement sur* $[0, T]$}

défini par :

$$\widetilde{\otimes}_T(\tau_1, \ldots, \tau_n) \equiv \otimes_T(\tau_{i,n_i}^{d_i})_{1\leq i\leq m}$$

L'exemple de la figure 4.5 montre un arbre d'ordonnancement construit à partir de deux tâches ayant chacune deux comportements d'exécution. Nous constatons bien que chaque noeud de l'arbre d'ordonnancement possède soit deux fils lorsque l'instruction exécutée est une instruction conditionnelle, soit un seul fils lorsque l'instruction n'est pas de type conditionnel ou soit aucun fils lorsque la profondeur de l'arbre correspond à la durée T de la construction de l'arbre d'ordonnancement.

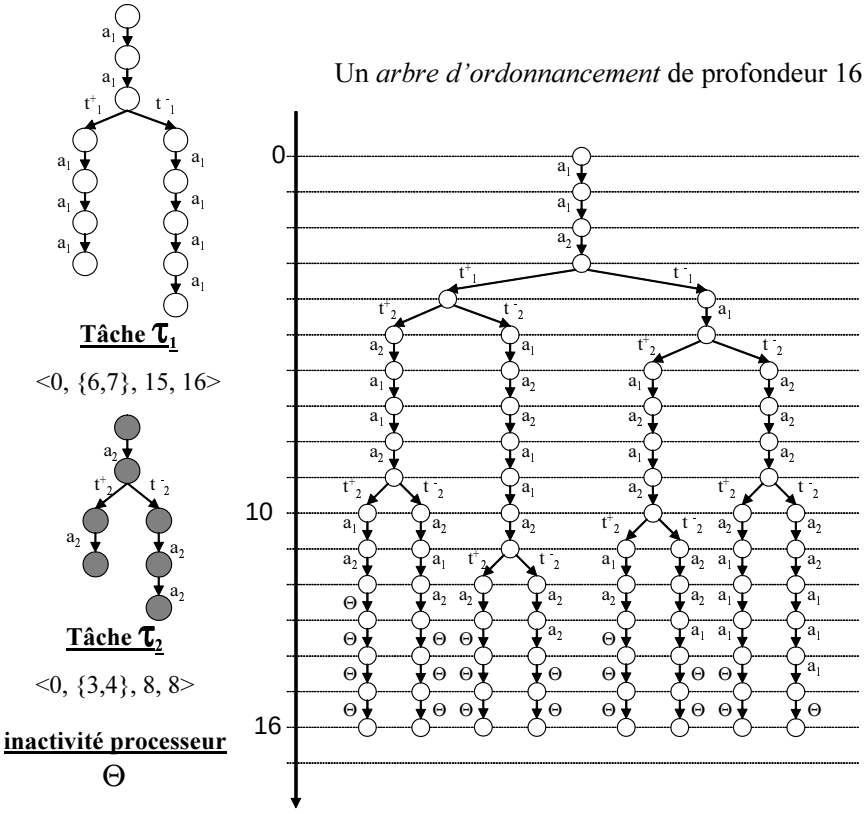

Fig. 4.5: Exemple d'arbre d'ordonnancement obtenu à partir des deux tâches $\tau_i \langle 0, \zeta_1 = \{7, 8\}, 16\rangle$ et $\tau_2 \langle 0, \zeta_2 = \{3, 4\}, 8, 8\rangle$ pour T=16. Nous pouvons noter l'ajout de Θ pour signifier l'inactivité processeur pendant une unité de temps.

L'opérateur $\widetilde{\otimes}_T$ génère tous les arbres d'ordonnancement conservatifs sur une fenêtre $[0, T]$ à partir d'un ensemble d'instances de tâches. Néanmoins, le respect des contraintes temporelles n'est pas encore vérifié. Il nous faut maintenant caractériser des arbres d'ordonnancement valides, c'est à dire qui ne transgressent pas les contraintes temporelles.

4.3.3 Arbre d'ordonnancement valide

Les opérateurs de génération de la définition 24 prennent en compte les dates de réveil r_i, les périodes P_i et les multi-ensembles ζ_i directement durant la phase de construction. Mais, nous n'avons pas jusqu'à présent tenu compte du partage des ressources ni des

contraintes temporelles des tâches. Il nous reste donc à intégrer la prise en compte des délais critiques D_i et la gestion des ressources.

Afin de caractériser ces arbres, nous introduisons pour tout alphabet $\alpha\beta_i$, l'alphabet barré

$$\overline{\alpha\beta_i} = \overline{a_i}, \overline{t^+_{i,1}}, \overline{t^-_{i,1}}, \ldots, \overline{t^+_{i,k}}, \overline{t^-_{i,k}}.$$

Cet alphabet permet de marquer durant la construction les étapes clés pour la vérification temporelle. Nous remplaçons dans les arbres d'exécution des tâches, les étiquettes des derniers liens par leur homologue dans $\overline{\alpha\beta_i}$. L'idée ici est de vérifier que la date d'apparition d'un élément de $\overline{\alpha\beta_i}$ est valide. Autrement dit, l'élément barré représentant la terminaison de chaque instance de tâche, nous pouvons par ce biais vérifier le respect des échéances de chaque tâche.

Pour la prise en compte des ressources, nous introduisons deux instructions $P_{Res_{i,j}}$ et $V_{Res_{i,j}}$ permettant de caractériser la prise et la libération de la ressource Res_i par la tâche τ_j. Celles-ci vont remplacer la lettre a_j dans l'alphabet $\alpha\beta_j$ lorsque la tâche τ_j prend ou rend une ressource. Nous associons également à chaque ressource Res_i, le nombre d'instances totales $Nb(Res_i)$ pouvant être prises simultanément. Ainsi $\forall\ Res_i$, nous avons $dispo\,(Res_i) = Nb(Res_i)$ à l'état initial (elles sont toutes libres), et les instructions $P_{Res_{i,j}}$ et $V_{Res_{i,j}}$ modifient respectivement en décrémentant et incrémentant de 1 le nombre d'instances disponibles de la ressource Res_i quelque soit la tâche τ_j. Ceci permet de prendre en compte les ressources multi-instances. Nous considérons que chaque tâche est correctement implémentée et par extension que l'arbre d'exécution associé garantit une bonne utilisation des ressources (par exemple, une ressource est prise avant d'être libéré, nous libérons bien toutes les instances prises *etc...*). En effet, nous ne faisons pas ici de test de validation structurel du code mais de la modélisation de code correctement implémenté. Nous pouvons noter toutefois que pour ce qui est des ressources nous pourrions nous assurer de leur bonne utilisation en vérifiant que chaque branche des arbres d'exécution de chaque tâche correspond à un langage de Dyck restreint et ce pour chaque ressource.

Les modifications que nous venons d'apporter sont intégrées lors de la construction des arbres d'ordonnancement : les opérateurs $\otimes_T$ et $\widetilde{\otimes_T}$ sont étendus et les lettres barrées sont gérées comme les lettres non bornées et les instructions de gestion de ressource comme les a_i.

Parmi les arbres produits par ces opérateurs, certains correspondent à des situations indésirables :

- soit certaines instances de tâches ratent leurs échéances. Une lettre barré survient après l'échéance de la tâche dont elle symbolise la terminaison. C'est une faute temporelle.
- soit les règles d'accès aux ressources critiques ne sont pas respectées. Les prises et libérations de ressources engendrent un nombre d'instances des ressources impossibles, c'est à dire $dispo\,(Res_i) > Nb(Res_i)$ *ou* $dispo\,(Res_i) < 0$.

Ces arbres sont dits non valides. Si par contre toutes les règles sont vérifiées, nous dirons que l'arbre est valide. La propriété suivante caractérise les arbres valides :

Propriété 4

Soit $\mathcal{A}$ un arbre obtenu grâce à l'opérateur $\widetilde{\otimes_T}$. On note p_h un noeud de profondeur h et T la profondeur de $\mathcal{A}$. L'arbre $\mathcal{A}$ est valide *si et seulement si pour toute branche $\mathcal{B}$ de $\mathcal{A}$ et pour toute ressource Res_u :*

1. *$\forall$ la tâche τ_j et $\forall k \in \{0, \ldots, \left\lfloor \frac{T-r_j}{P_j} \right\rfloor\}$ alors :*

$$\|\{l \setminus lien\ p_i \xrightarrow{l} p_{i+1}\ \in \mathcal{B},\ r_j + kP_j \leq i < r_j + kP_j + D_j, l \in \overline{\alpha\beta_j}\}\| = 1$$

2. *$\forall\ t \in [0, T]$ alors en notant*

$$\mathcal{M}_t = \|\{l \setminus lien\ p_i \xrightarrow{l} p_{i+1}\ \in \mathcal{B}\ avec\ 0 \leq i < t\ tel\ que\ \exists \tau_j\ l = P_{Res_{u,j}}\}\|$$

$$\mathcal{N}_t = \|\{l \setminus lien\ p_i \xrightarrow{l} p_{i+1}\ \in \mathcal{B}\ avec\ 0 \leq i < t\ tel\ que\ \exists \tau_j\ l = V_{Res_{u,j}}\}\|$$

alors

$$0 \leq \mathcal{M}_t - \mathcal{N}_t \leq Nb(Res_u)$$

Le point 1 permet de garantir que chaque instance de chaque tâche termine son exécution avant son échéance. En effet, pour chaque instance de tâche, nous regardons si la position de la dernière instruction marquée par l'alphabet barré $\overline{\alpha\beta_j}$ est prise en compte avant son échéance. Ainsi toutes les contraintes temporelles sont respectées de par l'opérateur de génération $\widetilde{\otimes}_T$ pour les dates de réveil et la période, et par cette propriété pour l'échéance. Le point 2, garantit quant à lui, le fait qu'une ressource ne peut être prise s'il n'y a plus d'instance libre. Pour cela, nous vérifions qu'à chaque instant et pour toute ressource Res_u, la différence entre le nombre d'instances prises $\mathcal{M}$ et le nombre d'instances libérées $\mathcal{N}$ est bien inférieure ou égale au nombre d'instances maximales de la ressource $Nb(Res_u)$.

Ces propriétés permettent de filtrer l'ensemble des arbres d'ordonnancement produits par l'opérateur $\widetilde{\otimes_T}$, afin de ne garder que les *arbres d'ordonnancement valides*, qui vérifient les contraintes temporelles de toutes les tâches et le partage de ressources pour chaque instance.

4.4 Ordonnançabilité Locale et Globale

Nous disposons maintenant d'un cadre formel des descriptions des comportements possibles d'une application. Il nous reste à redéfinir le problème de l'ordonnançabilité. De façon générale, il s'agit, dans le cas des durées fixes, de décider s'il existe ou non des séquences d'ordonnancement valides. Ici, nous pouvons envisager, soit une approche globale, soit une approche décompositionnelle.

4.4.1 Ordonnançabilité globale

Il s'agit de l'adaptation directe de l'ordonnançabilité dans le cas des durées fixes. Nous vérifions que l'opérateur $\widetilde{\otimes_T}$ produit au moins un arbre d'ordonnancement valide.

Définition 25 ***Ordonnançabilité globale***

Une application Temps Réel est dite globalement ordonnançable *sur une fenêtre d'observation T s'il existe un arbre d'ordonnancement valide de profondeur T.*

La notion d'ordonnançabilité globale indique que nous pouvons garantir les respect des échéances sans présager des résultats des tests à venir. Quelque soit le résultat d'un test, par définition et construction de l'arbre d'ordonnancement, il existe une branche de l'arbre ordonnançant le comportement adéquat. La figure 4.6 montre que l'application de l'exemple introductif 4.1 est globalement ordonnançable. En effet, nous pouvons exhiber un arbre d'ordonnancement valide sur la fenêtre de simulation $[0, 32]$.

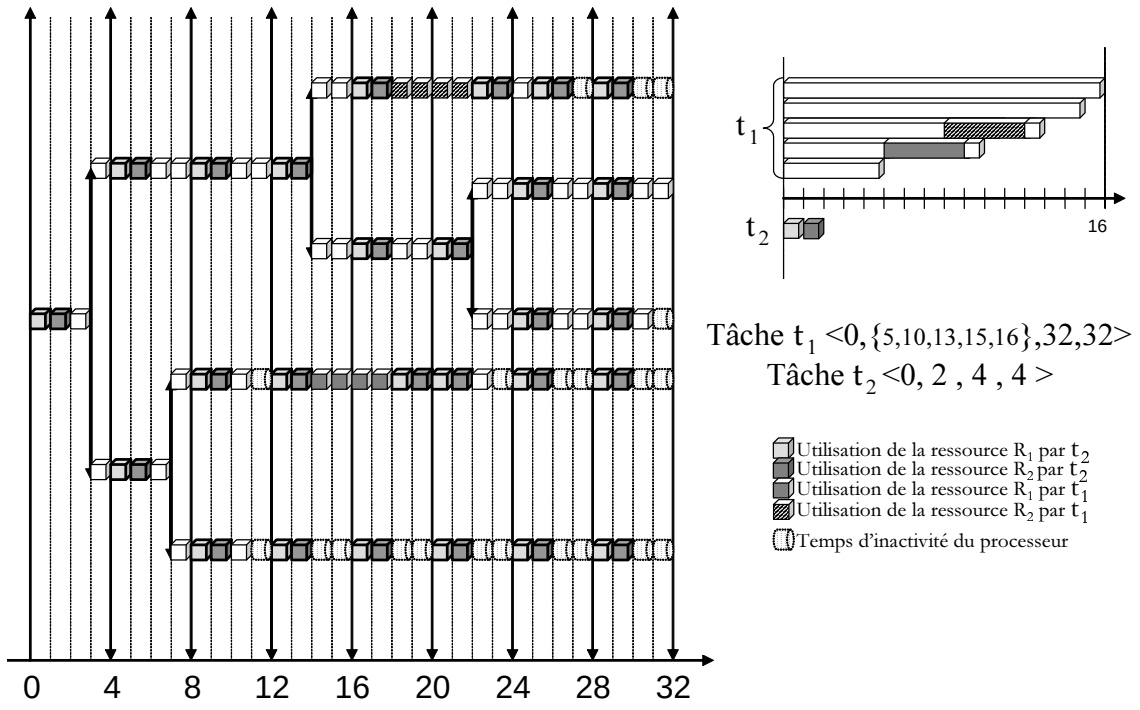

Fig. 4.6: Arbre d'ordonnancement pour l'application temps réel donné figure 4.1. Cette application est globalement ordonnançable sur $[0, 32]$.

4.4.2 Ordonnançabilité locale

Cependant, compte tenu des nombreux résultats existant pour les tâches à durées fixes, il peut être intéressant d'examiner individuellement les comportements possibles

de l'application de façon à ne pas considérer simultanément les deux alternatives d'un test. C'est ce que nous avons fait instinctivement dans l'exemple de 4.1 sur la figure 4.3. Nous avons voulu vérifier que chaque comportement d'exécution de l'application était effectivement ordonnançable. Nous formalisons dans un premier temps ce que sont les comportements possibles de l'application en introduisant la notion de configuration :

Définition 26 ***Notion de configuration***

– *Soit T une durée et τ_i une tâche , nous notons $e_i = \left\lceil \frac{T-r_i}{P_i} \right\rceil$ le nombre d'instances de τ_i activées dans la fenêtre d'observation $[0,T]$. Pour une tâche τ_i quelconque, nous définissons*

$$\mathcal{C}omp_{\tau_i} = \{\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,\wp_i}\} \text{ avec } \wp_i \text{ nombre de comportements de } \tau_i$$

l'ensemble de ses comportements*, c'est à dire l'ensemble de ses chemins d'exécution.*

– *Soit $(\tau_1, \ldots, \tau_n)$ une famille de tâches, nous définissons une* configuration $\mathcal{C}$ *pour une fenêtre $[0,T]$ par :*

$$\mathcal{C}_T = \{\tau^{(1)}_{1,j^1_1}, \tau^{(2)}_{1,j^1_2}, \ldots, \tau^{(e_i)}_{1,j^1_{e_i}}, \ldots, \tau^{(1)}_{n,j^n_1}, \ldots, \tau^{(e_n)}_{n,j^n_{e_n}}\}$$

avec $\tau^{(l)}_{i,j^i_l} = l^{ème}$ instance de τ_i qui s'exécute avec le comportement $\tau_{i,j^i_l} \in \mathcal{C}omp_{\tau_i}$

L'ensemble des configurations représente l'ensemble de tous les comportements possibles de l'application sur la fenêtre $[0,T]$, c'est à dire toutes les évolutions possibles de l'ensemble des tâches de l'application suivant les différents tests. Comme nous connaissons le nombre de comportements $\wp_i$ et le nombre d'instance e_i de chaque tâche τ_i, nous pouvons en déduire le nombre de configuration sur une fenêtre temporelle $[0,T]$.

Propriété 5 ***Nombre de configurations***

Le nombre de configurations sur $[0,T]$ est égal à

$$N = \prod_{i=1}^{n} (\wp_i)^{e_i}$$

Á partir de la notion de configuration, nous pouvons définir l'ordonnancement local qui consiste en l'analyse individuelle de chaque cas de figure. En effet, nous ne sommes alors en présence que d'instances de tâche à durée unique, mais pas forcément identique d'une instance à l'autre. Cette description de tâche nous ramène à l'étude d'un modèle de tâche multiframe [MC96](*cf* 1.3.1.3) avec les notions classiques d'ordonnaçabilité. Ainsi, une

configuration est ordonnançable si et seulement si il existe une séquence d'ordonnancement valide pour cette configuration.

Définition 27 *Ordonnançabilité locale*

Une application Temps Réel est localement ordonnançable *si chacune des N configurations est ordonnançable.*

L'ordonnançabilité locale, de par l'utilisation de la notion de configuration, anticipe sur les différents tests conditionnels. Cette stratégie clairvoyante est plus simple à mettre en oeuvre mais malheureusement elle ne pourra pas être appliquée dans le cas général, c'est à dire lorsque les tâches peuvent interagir (précédence, partage de ressources).

Nous pouvons toutefois constater une corrélation entre les notions d'ordonnançabilité locale et globale. En effet, chaque branche d'un arbre d'ordonnancement correspond à un ordonnancement d'une configuration et deux branches différentes correspondent à des configurations différentes. De plus pour être valide, l'arbre d'ordonnancement doit tenir compte de tous les comportements d'exécution de chaque tâche. Il y a donc bijection entre l'ensemble des branches et l'ensemble des configurations. Puisque l'arbre d'ordonnancement est valide, cela signifie que chacun des N ordonnancements des N configurations est valide. Nous venons donc de prouver la propriété suivante :

Propriété 6

L'ordonnançabilité globale implique l'ordonnançabilité locale sur les mêmes fenêtres d'observation.

Dans le cas où les tâches sont indépendantes, la réciproque est vraie. Mais elle ne l'est plus dans le cas général, en particulier lorsque les tâches utilisent des ressources dans les branches conditionnelles. C'est ce qu'illustrent les figures 4.7 et 4.8.

Les tâches de cet exemple sont à départs simultanés ($r_1,\ r_2,\ r_3 = 0$). Il suffit d'étudier le comportement de l'application sur la fenêtre [0,48] : nous avons $e_1 = e_3 = 1$ et $e_2 = \left\lceil \frac{48-0}{8} \right\rceil = 6$. Il y a donc 6 instances de τ_2 et τ_2 admet 2 comportements. τ_1 et τ_2 partagent la ressource 1, τ_2 et τ_3 la ressource 2. Nous pouvons ordonnancer localement chacune des $N = 1^1 \times 2^6 \times 1^1 = 64$ configurations comme sur la figure. Par contre, nous ne pouvons pas ordonnancer globalement ce système.

Considérons les 16 premières unités de temps : il y a 2 instances de τ_2, qui occupent le processeur 10 unités de temps, les 6 autres devront être utilisées par τ_1 et/ou τ_3 (comme U=100% et que les tâches sont à départs simultanés, il n'y a aucun temps creux [PC02]). Donc quand la $2^{ème}$ instance de τ_2 est réactivée au temps 8, $\tau_1 \cup \tau_3$ ont été exécutées durant 3 unités de temps, l'une d'entre elle a donc mobilisé l'une des 2 ressources. Par conséquent, si τ_2 s'exécute en prenant le chemin d'exécution correspondant à la ressource déjà prise, la tâche mobilisant la ressource doit la libérer avant que τ_2 ne puisse s'exécuter. Cela prendra au moins 4 unités de temps, ce qui empêchera τ_2 de respecter son échéance.

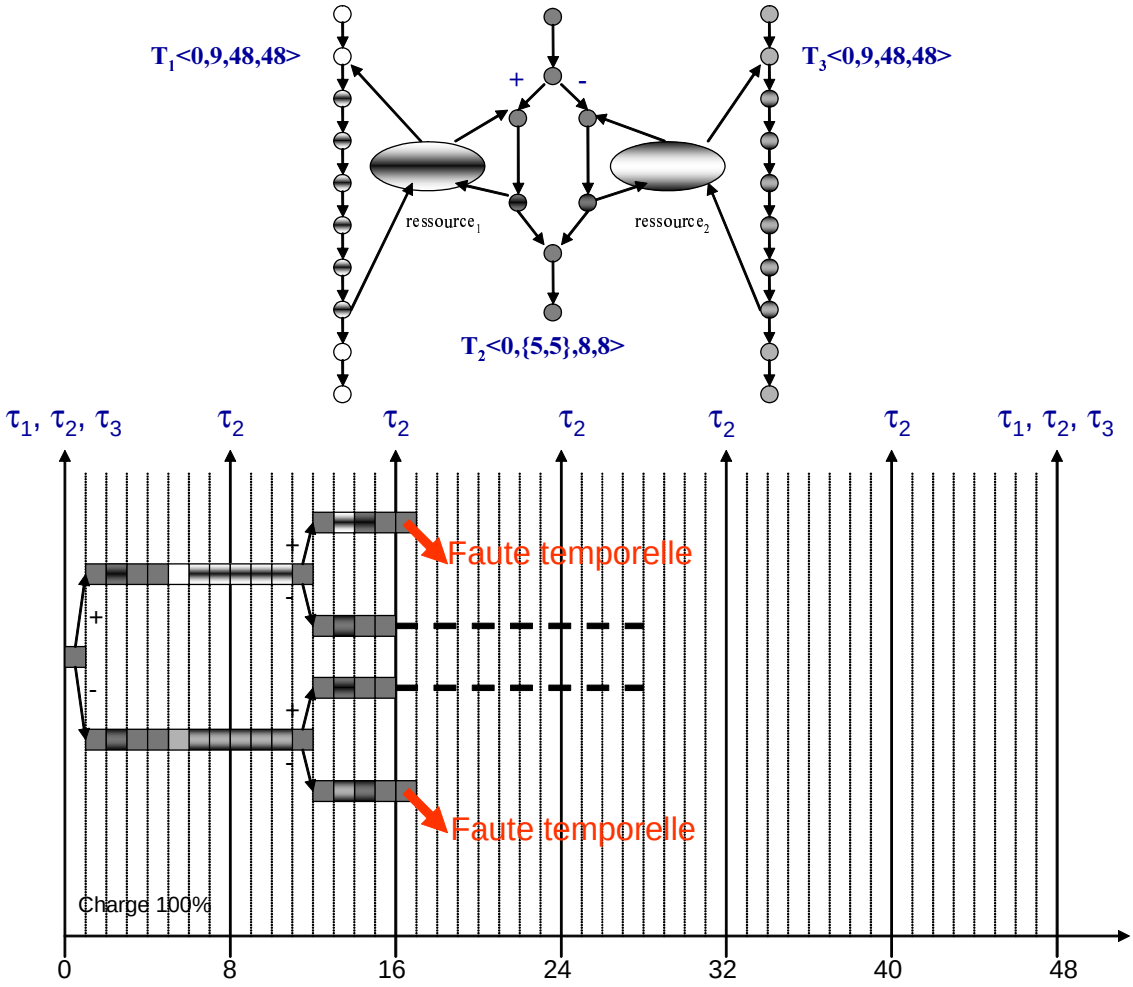

Fig. 4.7: Contre-exemple de la réciproque de la propriété 6 : l'ordonnançabilité locale n'est pas équivalente à l'ordonnançabilité globale dans le cas d'applications partageant des ressources. Nous pouvons constater que la construction d'un arbre d'ordonnancement est impossible

Par conséquent, cette application n'est pas globalement ordonnançable puisque nous ne pouvons pas construire d'arbre d'ordonnancement valide.

4.4.3 Cas des tâches indépendantes

Nous avons vu précédemment que la présence de ressources engendrait la non équivalence entre l'ordonnançacilité locale et globale. Nous montrons maintenant que dans le cas des tâches indépendantes, il y a bien équivalence entre ces deux notions.

Propriété 7 ***Équivalence***

Si nous considèrons un ensemble de tâches indépendantes, les deux notions d'ordonnançabilité sont équivalentes.

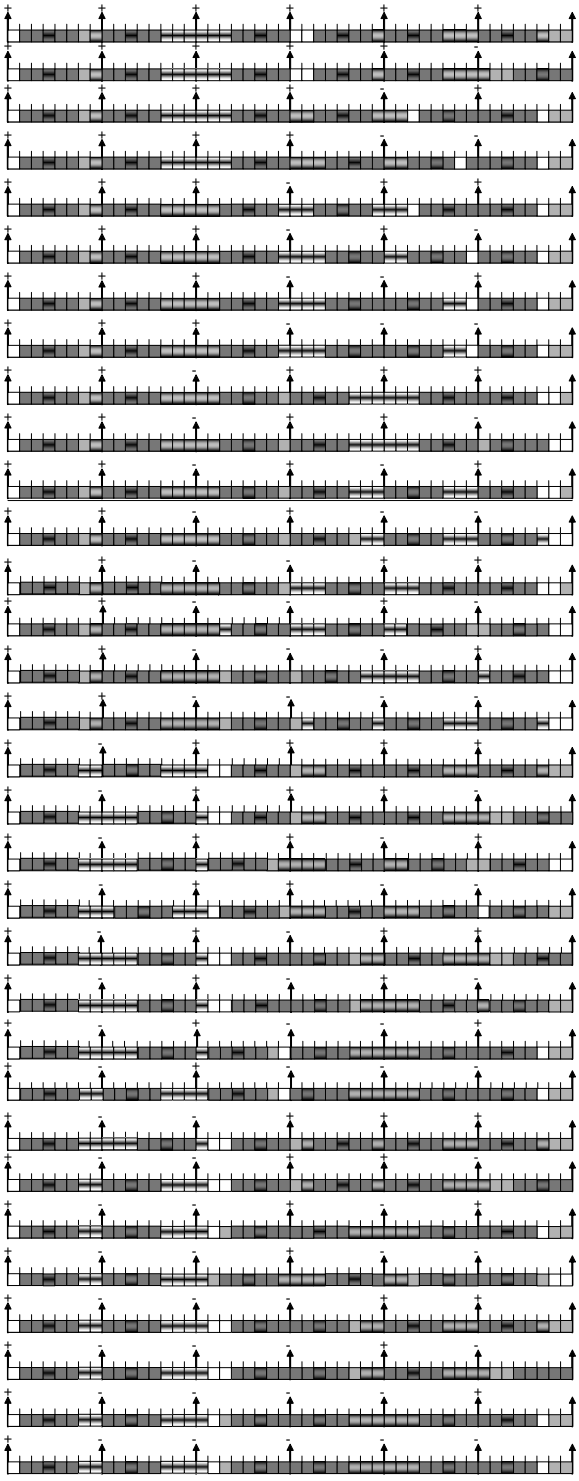

Fig. 4.8: L'application de l'exemple 4.7 est localement ordonnançable : il faut pour cela regarder l'ordonnançabilité des 64 configurations de l'application. Nous n'en montrons que 32 puisque les séquences restantes s'obtiennent par symétrie entre tâches 1 et 3. Les signes +et - indique le résultat du test conditionnel de la tâche 2 pour l'instance en cours.

PREUVE : Nous voulons montrer que l'ordonnançabilité locale implique l'ordonnançabilité globale dans le cas de tâches indépendantes. Considérons la configuration $\mathcal{C}_{Max}$ où chaque instance de chaque tâche adopte un comportement correspondant à la plus longue durée ($max(\zeta)$). Soit θ_T une séquence valide sur la fenêtre $[0, T]$ pour la configuration $\mathcal{C}_{Max}$. Nous savons qu'une telle séquence existe puisque l'application est localement ordonnançable.

Nous supposons que le système de tâches $(\tau_1, \ldots, \tau_n)$ comporte au moins une instruction conditionnelle. En isolant cette instruction, c'est à dire en repérant les lettres t^+ et t^- de l'alphabet de la tâche concernée, nous pouvons écrire la séquence comme suit :

$$\theta_T = m_0 t_1^+ m_1$$

avec m_0 une sous séquence d'ordonnancement θ_T ne comportant pas de test conditionnel. Sans perte de généralité, nous pouvons supposer que tous les résultats des tests correspondant à $\mathcal{C}_{Max}$ proviennent des alternatives positives.

Considérons que t_1^+ appartient à $\tau_i^{(\delta_i)} = \chi_1 t_1^+ \chi_1^+$. Soit $\tau_i^{(\delta_i')} = \chi_1 t_1^- \chi_1^-$ un autre comportement de $\tau_i^{(\delta_i)}$ tel que la durée de χ_1^- (noté $\|\chi_1^-\|$) soit maximale. Nous avons alors par construction $\|\chi_1^+\| \geq \|\chi_1^-\|$. Dans m_1, nous substituons χ_1^+ par $\chi_1^- \cdot \Theta^{\|\chi_1^+\| - \|\chi_1^-\|}$ d'où m_1' et nous construisons ainsi l'arbre

$$\theta_T = m_0 t_1 (m_1, m_1')$$

Ensuite, nous recommençons avec m_1 et nous ferons de même avec m_1'. Posons $m_1 = m_2 t_2^+ m_3$ avec t_2^+ appartient à $\tau_j^{(\delta_j)} = \chi_2 t_2^+ \chi_2^+$. Nous remplaçons de la même façon $\tau_j^{(\delta_j)}$ par $\tau_j^{(\delta_j')} = \chi_2 t_2^- \chi_2^-$ où χ_2^- est choisi de taille maximale. Nous substituons dans m_3, χ_2^+ par $\chi_2^- \Theta^{\|\chi_2^+\| - \|\chi_2^-\|}$ d'où m_3'. Nous construisons alors l'arborescence :

$$\theta_T = m_0 t_1 (m_2 t_2 (m_3, m_3'),\ m_2' t_2' (m_4, m_4'))$$

Et nous procédons de la même façon avec m_3, m_3', m_4, m_4' etc.... Comme le nombre total de tests est borné, le procédé termine.

Les substitutions qui sont effectuées respectent le point 1 de la propriété 4 quand à la position des lettres de l'alphabet $\overline{\alpha\beta} \cup \alpha\beta$ formant les tâches. En effet, comme avant la substitution il n'y avait pas de lettres de $\overline{\alpha\beta_i} \cup \alpha\beta_i$ entre $r_i + kP_i + D_i$ et $r_i + (k+1)P_i$, il ne peut y en avoir après substitution. En effet, l'opération de substitution enlève des lettres de $\overline{\alpha\beta_i} \cup \alpha\beta_i$ mais n'en rajoute jamais, ce sont des temps creux Θ que nous rajoutons éventuellement.

De même pour la date d'activation r_i, les lettres de $\overline{\alpha\beta_i} \cup \alpha\beta_i$ n'étant pas présentes avant r_i, il ne peut y en avoir après substitution. La position finale des lettres est un sous-ensemble des positions initiales ce qui permet de s'assurer également du respect de l'échéance D_i par les lettres de $\overline{\alpha\beta_i}$

L'arbre finalement construit respecte donc toutes les contraintes de la propriété 4. Tous les tests sont bien explorés, les branches de l'arbres sont toutes valides et de longueur T du fait des substitutions. Nous avons donc bien construit un arbre d'ordonnancement valide à partir d'une application localement ordonnançable ce qui prouve l'équivalence entre les ordonnançabilités globale et locale dans le cas de tâches indépendantes.

□

L'ordonnançabilité d'applications Temps Réel comportant des tâches indépendantes avec la considération des pires durées s'opère en un temps polynomial comme nous l'avons au chapitre 2, par conséquent il est plus intéressant de valider des applications comportant des tâches conditionnelles en considérant uniquement l'ordonnançabilité locale. De plus RM et ED restent optimaux dans leurs contextes respectifs pour l'ordonnançabilité globale puisqu'ils le sont pour l'ordonnançabilité locale.

Chacune des configurations, si elle est ordonnançable, l'est par RM ou ED dans le contexte des tâches indépendantes à départs simultanés. Donc, si l'application est ordonnançable globalement, elle l'est par ED ou RM.

Le dernier point à examiner est la valeur T de durée de simulation : les opérateurs $\widetilde{\otimes_T}$ et $\otimes_T$ construisent des arbres de profondeur T, cela permet d'étudier l'oronnançabilité sur la fenêtre $[0, T]$. Il nous reste à définir T de façon à ce que nous puissions conclure à l'ordonnançabilité globale ou locale en régime permanent (ie. sans limite de temps).

4.5 Durée de simulation et début de cyclicité

Nous avons redéfini le problème de l'ordonnançabilité en mettant en avant deux approches : l'ordonnançabilité locale et globale. Or, pour permettre de valider ces stratégies nous devons déterminer la fenêtre temporelle minimale sur laquelle la validation induit la garantie de l'absence définitive de faute temporelle. De plus, cela nous permettra de borner la profondeur de l'arbre à construire, si nous sommes dans une démarche hors-ligne, et nous indiquera comment obtenir le fonctionnement en régime permanent.

Nous pouvons d'ores et déjà constater que dans le cas de tâches à départs simultanés, une étude sur le Plus Petit Commun Multipe (PPCM) des périodes des tâches, noté P, est suffisante puisque toutes les tâches se retrouvent réactivées à l'instant P comme au temps 0. Il y a donc cyclicité des arbres d'ordonnancement sur une période P.

Pour rappel (*cf* 2.4) dans le cas de tâches à départs différés, modélisées par une stratégie du pire cas, les travaux de [CGG04] ont permis de déterminer une durée de simulation minimale permettant de valider les séquences d'ordonnancement en mettant en évidence la date de début de cyclicité. Ainsi, une séquence d'ordonnancement valide de type conservatif d'un ensemble de tâches $\langle \tau_1, \ldots, \tau_n \rangle$ à départs différés possède les propriétés suivantes :

- La séquence d'ordonnancement est cyclique et de période le PPCM des périodes de chacune des tâches τ_i.
- Le début de cyclicité a lieu au temps $t_c + 1$ où t_c est la date du dernier temps creux acyclique

Nous voulons montrer que ces résultats sont toujours vrais dans le cas de l'ordonnançabilité globale.

Propriété 8 ***Cyclicité des arbres d'ordonnancement***

Soit un système de tâches globalement ordonnançable. Quelque soit la stratégie d'ordonnancement de type conservatif choisie, les arbres d'ordonnancement sont cycliques et la date d'entrée dans le cycle est au plus égale à la date d'entrée dans le cycle lorsque nous considérons la configuration $\mathcal{C}_{Max}$.

PREUVE : Considérons un système de tâches indépendantes, à départs différés modélisées par le modèle temporel étendu. Supposons le système globalement ordonnançable. Quelque soit l'arbre d'ordonnancement, nous voulons montrer que :

- cet arbre est cyclique de période le PPCM (noté P) des périodes des tâches,
- cet arbre rentre en régime permanent au pire à la date t_c+1 avec t_c calculée à partir de la séquence d'ordonnancement correspondant à la configuration $\mathcal{C}_{Max}$.

Pour cela, nous ré-utilisons l'opération de substitution utilisée dans la preuve de la Propriété 7. Notons ainsi s une substitution élémentaire tel que s transforme le comportement d'une tâche issu d'un test t^+ par le comportement issu de l'alternative de ce même test t^- avec un comportement de durée maximale. Cet opération est déterministe puisqu'à un test donné, la transformation par substitution permet d'obtenir toujours le même comportement de durée max (dans le cas où il y a plusieurs comportements de même taille, nous choisissons toujours le même).

Soit $\mathcal{C}^{Max}_{[0,\ t_c+2\cdot P]}$ la configuration correspondant aux comportements de durée maximale pour chaque instance de tâches. Comme le système est globalement ordonnançable alors il existe une séquence d'ordonnancement de type conservatif pour $\mathcal{C}^{Max}_{[0,\ t_c+2\cdot P]}$, soit $\theta_{[0,\ t_c+2\cdot P]}$ cette séquence. Nous pouvons noter que $\theta_{[0,\ t_c+2\cdot P]}$ répond aux propriétés de cyclicité de [CGG04] puisque chaque tâche est ordonnancée avec une unique durée (celle correspondant au comportement de durée maximale) donc $\theta_{[0,\ t_c+2\cdot P]}$ est cyclique de période P avec t_c+1 le début de cyclicité. Nous pouvons ainsi noter :

$$\theta_{[0,\ t_c+2\cdot P]} = \theta_{[0,\ t_c]} \cdot \theta_{[t_c+1,\ t_c+P]} \cdot \theta_{[t_c+P+1,\ t_c+2\cdot P]}$$

Considérons maintenant une configuration $\mathcal{C}$ sur $[0,\ t_c + 2 \cdot P]$ et essayons de passer de $\mathcal{C}^{Max}$ à $\mathcal{C}$ sur $[0,\ t_c + 2 \cdot P]$. Nous devons transformer itérativement l'alternative de certains test en l'alternative opposée. Ceci induit une suite $S = s_1 \cdot s_2 \ldots s_w$ de substitutions élémentaire. Nous pouvons décomposer S en $s_1 \ldots s_u$ qui concerne les instances actives dans $[0,\ t_c]$, $s_{u+1} \ldots s_v$ pour les instances de $[t_c + 1,\ t_c + P]$ et enfin $s_{v+1} \ldots s_w$ pour $[t_c + P + 1,\ t_c + 2 \cdot P]$. La séquence d'ordonnancement $\theta_{[0,\ t_c+2\cdot P]}$ est alors transformée par ces substitutions en :

$$\begin{aligned} S \cdot \theta_{[0,\ t_c+2\cdot P]} &= (s_1 \dots s_u)\,\theta_{[0,\ t_c]} \cdot (s_{u+1} \dots s_v)\,\theta_{[t_c+1,\ t_c+P]} \cdot (s_{v+1} \dots s_w)\,\theta_{[t_c+P+1,\ t_c+2\cdot P]} \\ &= \theta'_{[0,\ t_c+2\cdot P]} \end{aligned}$$

$$\text{avec } \theta'_{[0,\ t_c+2\cdot P]} \text{ qui ordonnance } \mathcal{C} \text{ sur } [0,\ t_c + 2 \cdot P]$$

Or, puisque $\theta_{[0,\ t_c+2\cdot P]}$ est cyclique à partir de t_c alors les séquences $\theta_{[t_c+1,\ t_c+P]}$ et $\theta_{[t_c+P+1,\ t_c+2\cdot P]}$ sont identiques puisque de longueur P, la période de cyclicité.

La configuration $\mathcal{C}_{[0,\ t_c+2\cdot P]}$ peut également être décomposée sur les différents intervalles comme suit :

$$\mathcal{C}_{[0,\ t_c+2\cdot P]} = \mathcal{C}_{[0,\ t_c]} \cup \mathcal{C}_{[t_c+1,\ t_c+P]} \cup \mathcal{C}_{[t_c+P+1,\ t_c+2\cdot P]}$$

Si nous voulons garantir la propriété de cyclicité, il faut que les comportements d'exécution des tâches soient identiques sur deux PPCM consécutifs. Supposons que :

$$\mathcal{C}_{[t_c+1,\ t_c+P]} = \mathcal{C}_{[t_c+P,\ t_c+2\cdot P]}$$

Puisque les substitutions s sont déterministes alors nous avons :

$$(s_{u+1} \dots s_v) = (s_{v+1} \dots s_w)$$

De plus, nous savons que $\theta_{[t_c+1,\ t_c+P]} = \theta_{[t_c+P+1,\ t_c+2\cdot P]}$ car $\theta_{[0,\ t_c+2\cdot P]}$, est cyclique à partir de t_c, nous pouvons donc en déduire que

$$(s_{u+1} \dots s_v)\,\theta_{[t_c+1,\ t_c+P]} = (s_{v+1} \dots s_w)\,\theta_{[t_c+P+1,\ t_c+2\cdot P]}$$

Donc là encore, le système est dans le même état à t_c+1 et t_c+P+1, d'où la propriété de cyclicité.

Dans le cas où les tâches peuvent partager des ressources, le raisonnement est identique en faisant abstraction des ressources. En effet, quelque soit l'arbre d'ordonnancement choisi, nous avons la propriété de cyclicité et de borne de début de cyclicité t_c. Ce résultat étant vrai quelque soit l'arbre d'ordonnancement, il l'est donc en particulier pour les arbres respectant les propriétés de partage de ressources (cf Propriété 4).

□

4.6 Durée de simulation et début de cyclicité de système de tâches à durée d'exécution variable

La propriété précédente engendre un résultat plus général. En effet, nous étudions dans ce chapitre le problème de l'ordonnancement de tâches modélisées par un modèle temporel étendu tenant compte explicitement des instructions conditionnelles présent dans le code des tâches. Or, en considérant que chaque unité de temps des durées d'exécution de chaque tâche peut être une instruction conditionnelle, nous pouvons modéliser une tâche en tenant compte de toutes les durées d'exécution possibles entre une valeur minimale et maximale. La propriété précédente affirme ainsi que la durée de simulation et le début de

cyclicité peuvent être établis à partir des durées d'exécution maximales de chaque tâche. Nous pouvons ainsi constater que la date de début de cyclicité, c'est à dire de début de régime permanent ne dépend que de la charge du processeur.

Nous pouvons reformuler la propriété précédente, étendue au cas général par le corollaire suivant :

Corollaire 1

Quelque soit les paramètres temporels r_i, D_i et P_i d'un ensemble de tâches, la date d'entrée dans le cycle t_c est une fonction décroissante au sens large de la charge du système.

Ce corollaire nous indique que l'analyse d'ordonnançabilitité d'un arbre d'ordonnancement peut s'effectuer sur une durée de simulation bornée d'après la cyclicité des arbres d'ordonnancement de la propriété 8, et que cette durée de simulation est décroissante et ne dépend que de la décroissance globale des durées d'exécution de l'ensemble des instances de chaque tâche. Par extension et comme toute tâche à durée variable peut être ramenée à une tâche conditionnelle, alors dans le cas général d'applications Temps Réel multi-tâches, une diminution de durée d'exécution d'une tâche sur au moins une instance ne peut pas faire augmenter la durée de simulation nécessaire à la validation temporelle de cette application.

4.7 Conclusion

Après avoir mis en évidence les difficultés d'ordonnançabilité que peut engendrer une modélisation temporelle de Liu-Layland d'un système de tâches partageant des ressources, nous avons proposé un nouveau modèle de tâche permettant de ne pas s'appuyer sur la considération du pire cas. Ainsi, si une tâche peut être exécutée avec n comportements d'exécution distincts issus de tests conditionnels, nous choisissons de prendre en compte explicitement ces n comportements dans notre modèle de tâches en intégrant chaque test conditionnel dans la représentation de la durées des tâches. Nous introduisons alors non pas une durée d'exécution C_i déterministe et évaluée en considérant le comportement d'exécution maximale de chaque tâche, mais un multi-ensemble de durées ζ représentant chaque comportement d'exécution d'une tâche.

Ce modèle de tâche étendu nous a obligés à revoir la notion d'ordonnancement, en introduisant la notion d'arbre d'ordonnancement. Nous proposons un opérateur de génération des arbres d'ordonnancement de type conservatif. Nous avons introduit des propriétés de vérification du respect des contraintes temporelles de chaque tâche et du partage des ressources pour permettre la construction d'arbres d'ordonnancement valides. Nous avons par la suite proposé deux stratégies d'ordonnancement : l'ordonnançabilité locale vérifie l'ordonnançabilité de chaque configuration d'exécution possible, pour un système de tâches en utilisant les techniques de l'ordonnancement classique. L'ordonnançabilité globale quant à elle repose sur l'existence d'un arbre d'ordonnancement valide. Nous avons

montré que, dans le cas de tâches indépendantes, ces deux stratégies sont équivalentes ce qui permet de profiter des résultats déjà établis dans le cas du modèle de tâche classique pour l'ordonnançabilité globale.

Enfin, nous avons montré que ces arbres d'ordonnancement pouvaient être étudiés sur une durée de simulation bornée pour leur validation. Nous avons donc mis en évidence la cyclicité des arbres d'ordonnancement en prouvant que la date de rentrée dans le cycle était décroissante au sens large lorsque les durées d'exécution sont globalement décroissantes.

Ces résultats permettent d'envisager des études d'ordonnançabilité plus précises en évitant de sur-contraindre la représentation du système de tâches. La prise en compte explicite des instructions conditionnelles permet d'envisager l'ordonnancement de systèmes fonctionnant sous plusieurs régimes (mode normal/mode dégradé) comme pour la gestion de l'économie d'énergie par exemple. Nous avons donc posé les bases formelles du problème de l'ordonnancement dans le contexte des applications à tâches conditionnelles.

Les techniques d'ordonnancement hors-ligne vont permettre de mettre en pratique les concepts développés. En particulier, il convient de développer des outils efficaces de génération des arbres d'ordonnancement, qui permettent une intégration dans le processus même de construction, de la vérification de la propriété 4 de validité. C'est précisément ce que nous allons voir au chapitre suivant, une méthode fondée sur une modélisation de l'application par réseau de Petri généralisant les travaux de [GCG02] vu au chapitre précédent, proposée dans [PC04].

4.8 Bibliographie

[CC91] H. Chetto and M. Chetto. An adaptative scheduling algorithm for fault-tolerant real-time systems. In *Software Engineering Journal*, pages 179–186, Mai 1991.

[CET01] Samarjit Chakraborty, Thomas Erlebach, and Lothar Thiele. On the complexity of scheduling conditional real-time code. *Proceedings of the Seventh International Workshop on Algorithms and Data Structures (WADS 2001)LNCS 2125*, pages 38–49, 2001.

[CGG04] A. Choquet-Geniet and E. Grolleau. Minimal schedulability interval for real time systems of periodic tasks with offsets. In *Theoretical of Computer Sciences*, volume 310, pages 117–134, March 2004.

[CHB79] R.H. Campbell, K.H. Hurton, and G.G. Belford. Simulations of fault-tolerant real-time deadlin mechanisms. In *Proceedings of FCTS'9*, pages 95–101, Janvier 1979.

[CL88] J.Y. Chung and J.W.S Liu. Algorithms for scheduling periodics jobs to minimize average error. In *9th IEEE RTSS*, pages 142–152, Décembre 1988.

[GCG02] Emmanuel Grolleau and Annie Choquet-Geniet. Off-line computation of real-time schedules using petri nets. In *Discrete Event Dynamic Systems, DEDS*, volume 12(3), pages 311–333. Kluwer Academic Publishers, Jully 2002.

[MC96] A. K. Mok and D. Chen. A multiframe model for real-time tasks. In *RTSS '96 : Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 22, Washington, DC, USA, 1996. IEEE Computer Society.

[PC02] S. Pailler and A. Choquet-Geniet. Ordonnancement temps réel comportant des tâches à durées variables. In Tecknea, editor, *RTS Embedded System*, pages 151–172, Paris, France, 2002.

[PC04] S. Pailler and A. Choquet-Geniet. Off-line scheduling of real time aplications with variable duration tasks. In Kluwer Academic Publisher, editor, *Workshop on Discrete Event Systems, WODES2004, Discrete Event Systems Analysis and Control*, Reims, France, September 2004.

[Ric02] Michaël Richard. *Contribution à la Validation des Systèmes Temps Réel Distribués : Ordonnancement à Priorités Fixes & Placement.* PhD thesis, LISI, ENSMA, Université de Poitiers, Novembre 2002.

Chapitre Cinq

Analyse d'applications comportant des tâches conditionnelles

Chapitre Cinq

Analyse d'applications comportant des tâches conditionnelles

Notre objectif est d'étudier l'ordonnançabilité d'une application temps réel pouvant être constituée de tâches à durées variables, plus précisément de tâches conditionnelles comme vu précédemment. Celles-ci peuvent être communicantes, partager des ressources en mode lecture écriture ou de façon exclusive, elles peuvent également posséder des blocs de durées non préemptibles. La méthode utilisée reprend celle de [Gro99] et consiste en une modélisation par RdP autonome coloré avec la règle de tir maximal. La prise en compte de tâches conditionnelles engendre la notion d'arbre d'ordonnancement. Nous devons par conséquent étendre la modélisation par RdP et permettre une analyse du graphe des marquages aboutissant à l'extraction d'arbres d'ordonnancement. Nous montrons également que l'utilisation de tâches conditionnelles permet de simuler le déclenchement sporadique d'une tâche tierce. Nous montrons comment intégrer ces tâches sporadiques déclenchées dans notre approche.

Sommaire

Chapitre Cinq

Analyse d'applications comportant des tâches conditionnelles

Afin de tenir compte des fluctuations de durée des exécutions des tâches, nous avons définis un nouveau modèle temporel de tâches qui étend celui issu de Liu Layland. Nous pouvons ainsi être plus réaliste du point de vue de l'abstraction du code des tâches pour la détermination des durées d'exécution et de l'utilisation des ressources. Notre objectif est maintenant de proposer une méthode d'analyse hors ligne, qui tienne compte des spécificités des tâches conditionnelles en s'appuyant sur les concepts mis en place dans le chapitre précédent.

Nous proposons donc une méthode qui étend celle de Grolleau [Gro99, GCG02] vue au chapitre 3 pour intégrer ce nouveau modèle de tâche. Nous rappelons que cette méthode est basée sur une modélisation par Réseaux de Petri autonomes colorés munis de la règle de tir maximal de l'application temps réel. Celle-ci peut être composée de tâches à départs différés ou simultanés, pouvant communiquer par rendez vous, partager des ressources en mode lecture/écriture ou en mode exclusif, exécuter des blocs de durée en mode non préemptif. Cette modélisation permet une énumération des solutions au travers de la construction du graphe des marquages associé. Ce dernier pouvant être restreint par l'utilisation de contraintes tant qualitatives (comme pour l'optimalité de certains critères) que quantitatives (comme le respect de contraintes absolues).

Nous présentons les implications de la prise en compte des tâches conditionnelles du point de vue de la modélisation par RdP ainsi que de la construction et l'exploitation du graphes des marquages. En ce qui concerne la modélisation, nous devons en particulier revoir les caractéristiques (et donc la modélisation) de la tâche oisive, puisque les fluctuations de durées impliquent une variation du nombre de temps creux. en ce qui concerne l'analyse du réseau, nous présentons une méthode d'extraction non plus de séquences, mais d'arbres d'ordonnancement. Nous pouvons noter que l'emploi de tâches conditionnelles engendre une augmentation non négligeable de la taille du graphe des marquages. Nous verrons alors quelques critères analytiques permettant de restreindre ce dernier aux seules informations pertinentes au regard des critères retenues.

Par ailleurs, nous nous penchons sur les possibilités qu'offrent le couplage de tâches conditionnelles avec l'utilisation des communications entre tâches. Le nouveau modèle de tâche permet d'avoir une vision structurelle des différentes exécutions d'une tâche, il est alors possible de faire en sorte que des communications entre tâches s'effectuent uniquement pour certains comportements d'exécution. Il est ainsi possible de simuler l'activation sporadique de tâches déclenchées par l'un des comportements d'exécution d'une

tâche conditionnelle. Intuitivement, lors de la conception d'une application temps réel, la présence de tâches sporadiques peut être vue comme une routine de traitement d'une situation ne faisant pas partie du régime permanent du système. Or, le déclenchement d'une telle routine peut provenir d'une tâche de contrôle vérifiant le bon fonctionnement du système. Nous retrouvons alors le processus de déclenchement de tâche par synchronisation sur des comportements d'exécution particulier d'une tâche conditionnelle.

Nous montrons comment modéliser les tâches sporadiques déclenchées et comment extraire des arbres d'ordonnancement incluant la gestion des sporadiques. Cette analyse permet ainsi de gérer hors ligne l'exécution de tâches sporadiques déclenchées par activations logicielles.

À notre connaissance, aucune approche hors ligne ne prend en compte un modèle de tâche aussi détaillé. Dans [LG02], une modélisation des comportements conditionnels est envisagée, mais la démarche proposée répond essentiellement au problème de l'ordonnançabilité (l'application est ordonnançable si et seulement si le centre du langage produit par l'automate construit est non vide). De plus, le problème des précédences (et par delà des tâches sporadiques déclenchées) n'est pas pris en compte. Dans [XP90] qui présente une méthode hors ligne prenant en compte ressources et précédences, l'optique adoptée est celle du pire cas. Enfin, dans [Ric02] les fluctuations de durées sont prises en compte mais d'une part, ce sont les stratégies à priorités fixes qui sont envisagées et d'autre part, les tests de validation sont des conditions suffisantes d'ordonnançabilité (fondée sur l'évaluation sur un majorant des pires temps de réponse).

Dans une première partie nous montrons comment modéliser par RdP des tâches conditionnelles. Nous regardons plus particulièrement les implications au niveau des ressources, des synchronisations et des temps creux acycliques permettant de gérer les tâches à départs différés. Nous montrons en particulier comment la fluctuation de durée des tâches conditionnelles induit la variation du nombre de temps creux ce qui a pour effet de faire de la tâche oisive une tâche à durée variable.

Dans une deuxième partie, nous montrons comment utiliser cette tâche oisive pour tenir compte des tâches sporadiques déclenchées. Dans un premier temps nous montrons comment modéliser par RdP ce type de tâches puis quelles en sont les implications structurelles.

Enfin, dans une dernière partie, nous étudions plus finement la construction du graphe des marquages et l'extraction d'arbres d'ordonnancement. Nous nous penchons sur l'accroissement de la complexité de la méthode, puis nous regardons comment construire un graphe d'accessibilité valide. Nous verrons alors comment extraire des arbres d'ordonnancement à l'aide de contraintes quantitatives et/ou qualitatives.

5.1 Modélisation des tâches conditionnelles

La méthode proposée pour étudier l'ordonnançablité d'une application temps réel est une adaptation de la méthode présentée dans [CGC96, GG00]. Elle consiste en une modélisation de l'application par un réseau de Petri, puis en une analyse de ce réseau par construction du graphe des marquages terminaux, et extraction des séquences valides. Nous cherchons à étendre cette méthode de façon à tenir compte des tâches conditionnelles et donc de l'ensemble de leurs comportements d'exécution. Nous avons vu précédem-

ment (chapitre 3) comment modéliser par RdP une application temps réel comportant des tâches à départs différés ou simultanés, partageant des ressources, pouvant communiquer et exécuter des blocs de durée non-préemptibles.

Nous rappelons que cette modélisation comporte deux parties :

- un réseau modélisant la structure temporelle. Nous utilisons un modèle discret du temps, une horloge externe (noté RTC) comptabilise chaque unité de temps correspondant à l'intervalle s'écoulant entre deux ticks successifs. Par ailleurs chaque tâche est munie d'une horloge locale permettant de comptabiliser le temps écoulé depuis la dernière activation de la tâche.
- un réseau modélisant le système de tâches. Chaque transition correspond à une action de durée une unité de temps et toutes les transitions sont en concurrence pour l'obtention du processeur. Les activations de la tâche sont modélisées par les jetons a et b. Les délais critiques sont pris en compte à l'aide d'ensembles terminaux.

5.1.1 Les tâches conditionnelles

La représentation par RdP d'une tâche conditionnelle est basée sur la représentation arborescente issue du modèle de tâche étendu du chapitre précédent. La modélisation des branches conditionnelles se fait de façon classique par deux transitions concurrentes t^+ et t^- correspondant chacune à une alternative du test conditionnel. Si la tâche se termine par un bloc de durée issu d'un instruction conditionnelle, chacune des branches doit se terminer par une transition qui dépose la marque b de terminaison dans la place $Active_i$. De cette façon, les ensembles terminaux vus au chapitre 3 et définis dans le cas déterministe (durée du pire cas) restent préservés comme nous pouvons le voir sur la figure 5.1.

Nous pouvons toutefois noter qu'il n'y a pas obligatoirement autant de transitions de terminaison (c'est à dire de transitions qui déposent sur jeton b dans la place $activ_i$) qu'il a de comportements d'exécution possibles. En effet, la modélisation par RdP de ce type de tâche respecte la structure grammaticale du pseudo code qui les décrit et non une énumération des différents comportements telle qu'une représentation en arbre binaire peut le faire. L'exemple de la figure 5.2 suivante illustre cette définition.

Nous considérons qu'un test conditionnel dure une unité de temps. En effet, lorsque le test est de durée supérieure, nous pouvons toujours nous ramener à une unité de temps en augmentant les blocs de durée suivants de la durée manquante. Nous pouvons noter également que les marquages terminaux vus au chapitre 4 pour les tâches à durée déterministe (vision du pire cas) s'appliquent de la même façon aux tâches conditionnelles.

5.1.2 La tâche oisive

Dans le modèle initial, la tâche oisive était modélisée comme toutes les autres tâches. Mais compte tenu de ses spécificités, la modélisation peut être revue et allégée. En effet, par définition du facteur de charge, nous savons qu'il y aura effectivement $P(1-U)$ temps d'inactivité du processeur toutes les méta-périodes P, ce qui garantit que la tâche oisive sera intégralement exécutée toutes les méta-périodes. Il est donc superflu de doter la tâche oisive d'un délai critique. L'usage des jetons a et b et de l'ensemble terminal concernant la

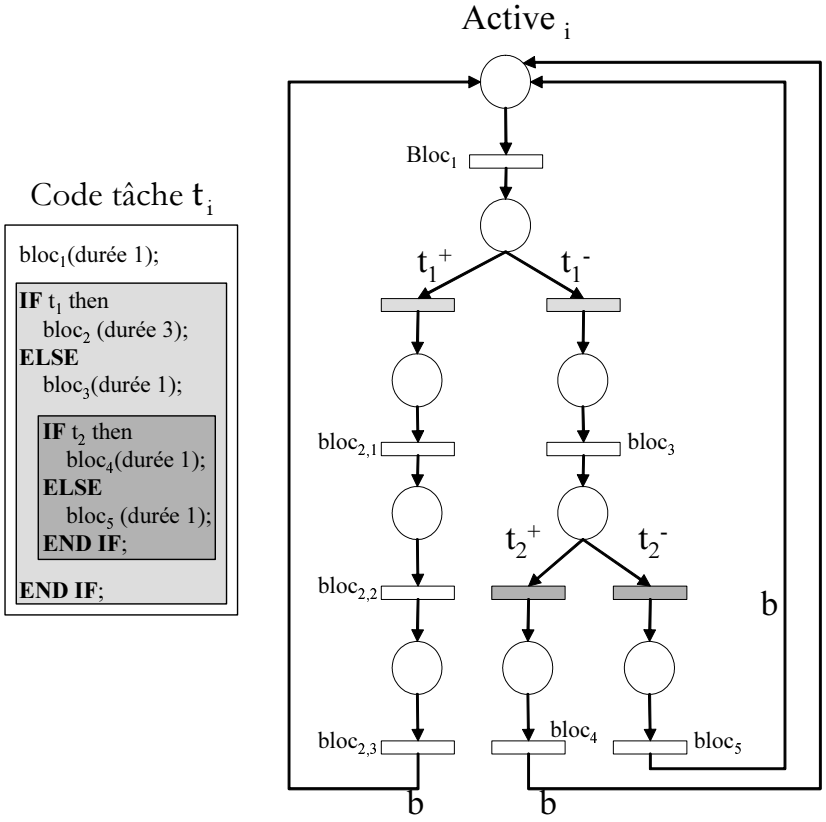

Fig. 5.1: Modélisation par RdP d'une tâche conditionnelle τ_i : les trois transitions terminales de la tâche déposent un jeton b dans la place $Active_i$.

tâche oisive devient également superflu. Cette dernière peut donc être modélisée à l'aide d'une seule place et d'une seule transition comme le montre la figure 5.3 : la transition $Activation_0$ dépose $P(1-U)$ marques dans $Active_0$ toutes les P unités de temps, marques qui seront consommées par la transition $Oisif_0$.

En présence de tâches conditionnelles, le facteur de charge U du système n'est plus déterministe, par suite, la durée de la tâche oisive $P(1-U)$ devient elle aussi non déterministe. Afin de préserver le fonctionnement sous la règle de tir maximal, il nous faut modéliser les fluctuations de la durée d'inactivité du processeur. Deux méthodes peuvent être envisagées, la méthode de la plus longue durée et la méthode de la plus courte durée :

· **Méthode de la plus longue durée**

Nous donnons par défaut à la tâche oisive sa durée maximale, obtenue à partir du facteur de charge minimal U_{min}. La transition $Activation_0$ dépose donc $P(1-U_{min})$ jetons dans la place $Activ_0$. Ceci revient à choisir par défaut dans chaque bloc conditionnel la branche de plus courte durée. Si l'alternative choisie à l'intérieur d'une tâche n'est pas celle induisant le traitement de plus courte durée, afin de ne pas dépasser le seuil de 100% de charge, il faut diminuer la durée de la tâche oisive, c'est à dire retirer des marques de la place $Active_0$, comme l'illustre la figure 5.4. L'inconvénient de cette approche est qu'il y a augmentation des retours en arrière

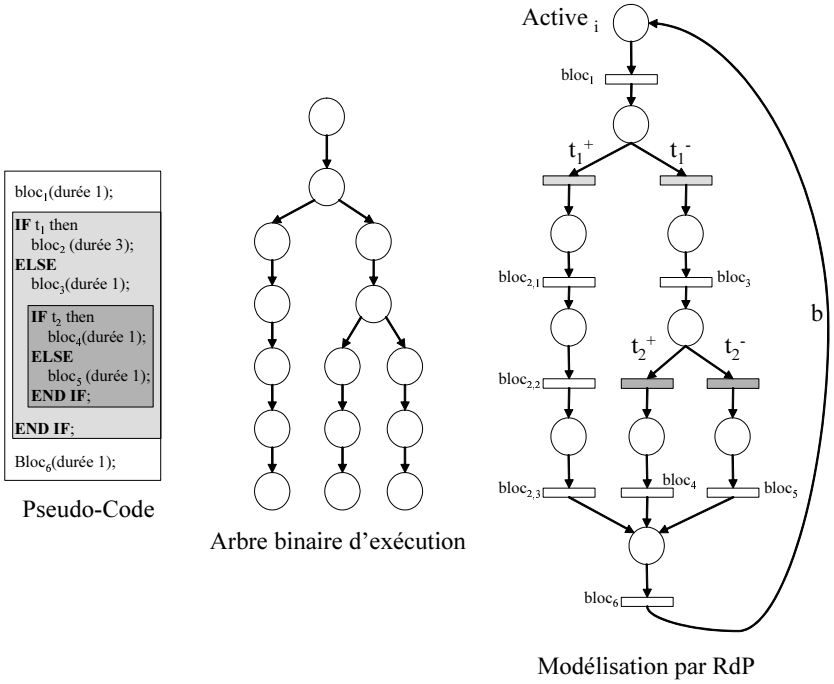

Fig. 5.2: Modélisation par RdP d'une tâche conditionnelle τ_i : une seule transition terminale et plusieurs comportements d'exécution.

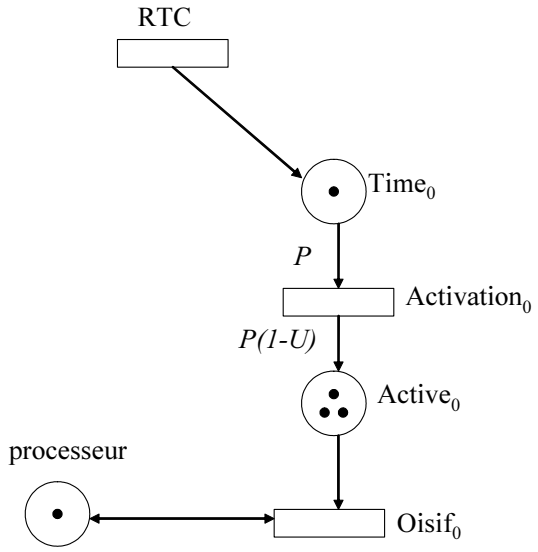

Fig. 5.3: Modification de la modélisation de la tâche oisive.

lors de la construction du graphe des marquages. En effet, si il y a eu trop de temps creux utilisés avant le choix, il se peut qu'il n'y ait plus assez de marques dans la place $Active_0$ pour permettre le choix d'une branche longue. Dans l'optique de l'étude de l'ordonnançabilité globale, une telle approche, qui permet l'anticipation sur des choix ultérieurs, n'est pas souhaitable ; par contre, elle devient nécessaire si nous désirons

faire une étude d'ordonnançabilité locale, et que nous souhaitons récupérer toutes les séquences valides pour chacun des comportements indépendamment des autres.

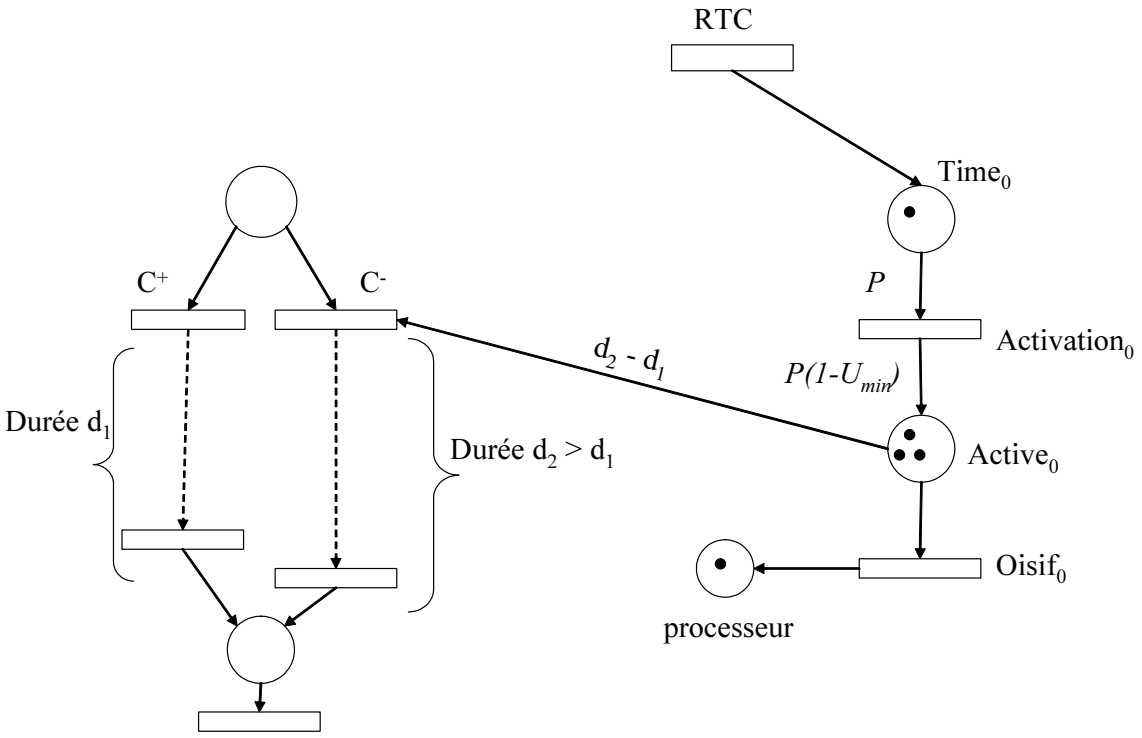

Fig. 5.4: Modélisation de la tâche oisive par la méthode de la plus longue durée.

· **Méthode de la plus courte durée**

Cette fois, nous utilisons la valeur maximale du facteur de charge U_{max}. Dans le cas où l'alternative choisie n'est pas celle induisant la plus longue durée, il faudra cette fois allonger la durée de la tâche oisive, donc rajouter des jetons dans la place $Active_0$. C'est ce qu'illustre la figure 5.5. Cette technique n'induit pas de retours en arrière, puisqu'il n'y a pas anticipation sur les choix faits. Elle s'adapte bien à l'étude de l'ordonnançabilité globale. C'est la méthode que nous avons retenue dans la suite.

5.1.3 Tâches à départs différés

Nous avons vu au chapitre 3 comment tenir compte des tâches à départs différées dans notre modélisation par RDP. Nous avons vu également que le cas de tâches à départs différés engendre l'apparition de temps creux acycliques dans les séquences d'ordonnancement. Ces temps creux acycliques sont pré-calculés par un diagramme de charge et peuvent ainsi être injectés dans les séquences d'ordonnancement de façon à ne pas produire uniquement des segments d'ordonnancement conservatifs, qui, ne sont pas optimaux dans le cas de tâches à départs différés partageant des ressources et/ou comportant des blocs de durée non préemptibles.

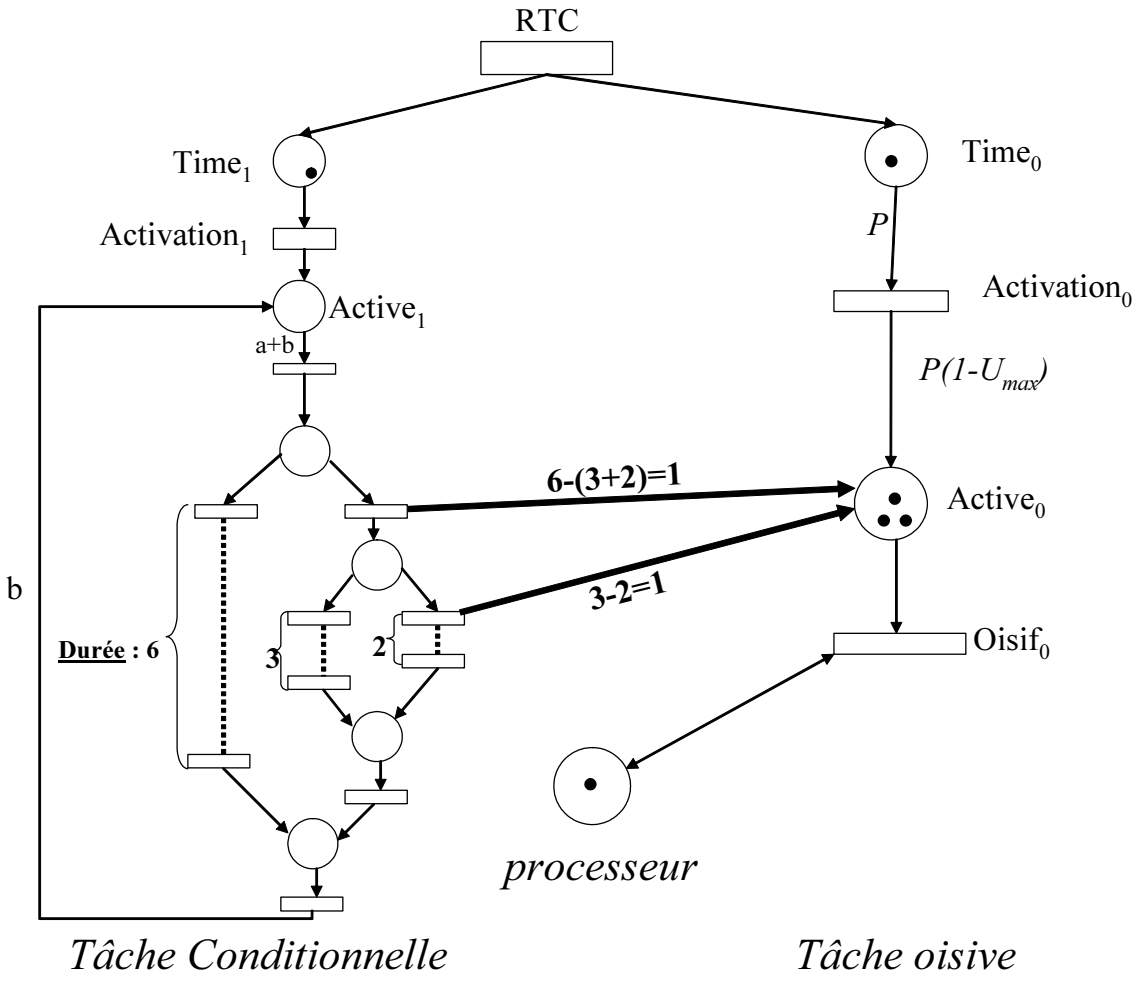

FIG. 5.5: Ajustement de la charge de la tâche oisive pour la méthode de la plus courte durée avec $U = 0,75$, $P = 20$ et donc $C_0 = 5$.

Dans le cas des tâches conditionnelles, la variation de la charge processeur dues aux fluctuations des durées d'exécution empêchent a priori d'employer la même méthode basée sur un diagramme de charges pour le calcul des temps creux acycliques. En effet, puisque la charge du système fluctue suivant les comportements d'exécution effectivement exécutés par le processeur, il n'est a priori pas possible de déterminer le nombre de temps creux acycliques.

Or, nous avons montré (Propriété 8) que dans le cas d'applications temps réel comportant des tâches conditionnelles, la date du dernier temps creux acyclique est une fonction décroissante au sens large de la charge processeur. Donc, en considérant les pires durées, nous pouvons donner un majorant de la date du dernier temps creux acyclique. De ceci découle déjà une borne de la profondeur du graphe des marquages à construire (c'est équivalent à la profondeur des arbres d'ordonnancement à produire). Ensuite, nous déterminons les temps creux acycliques toujours en considérant pour chaque tâche la pire durée d'exécution. Il ne reste plus qu'à étudier plus finement le segment allant jusqu'à la date au plus tard de dernier temps creux. Deux cas peuvent se produire,

- Soit durant l'exécution effective de l'application, il s'avère qu'un nombre plus important de temps creux acycliques est nécessaire. Dans ce cas, le fonctionnement par RdP engendrera naturellement les temps creux acycliques manquants puisqu'aucune tâche ne pourra s'exécuter par définition des temps creux acycliques.
- Soit durant l'exécution effective de l'application, il s'avère qu'il y avait trop de temps creux acycliques, ces derniers sont alors transformés en temps creux cycliques. En effet, s'il y a trop de temps creux acyclique cela signifie que la cyclicité a commencé avant la date prévue par le diagramme de charge des pires durées. Par conséquent, les temps creux acycliques en surplus sont intégrés à la tâche oisive.

La prise en compte des tâches conditionnelles à départs différés n'engendre donc aucun changement par rapport aux applications composées de tâches à durées du pire cas.

5.1.4 Gestion des Ressources

La prise en compte des ressources s'opère de la même façon que dans [Gro99]. Les instructions $Prendre\_Semaphore(Res_i, I_{nstance})$ et $Rendre\_Semaphore(Res_i, I_{nstance})$ une utilisation des ressources :

- en exclusion mutuelle en utilisant chaque ressource avec l'ensemble de leur instances c'est à dire avec $I_{nstance}= nb_{max}(Res_i)$
- en lecture/écriture, nous considérons que la lecture ne prend qu'une instance de la ressource ($I_{nstance} = 1$) ce qui permet $nb_{max}(Res_i)$ lectures simultanées. L'écriture nécessite toutefois l'ensemble des instances de la ressource ($I_{nstance}= nb_{max}(Res_i)$) pour s'assurer de l'exclusion mutuelle.

L'utilisation de tâches conditionnelles n'introduit aucun changement si ce n'est la possibilité d'utiliser des ressources uniquement pour certains comportements d'exécution. la figure 5.6 illustre le partage en Lecture/Écriture d'une ressource R par deux tâches τ_1 et τ_2, qui sont deux tâches conditionnelles, l'un des comportements de τ_1 utilise la ressource en lecture et l'un des comportements de τ_2 l'utilise en écriture.

5.1.5 Gestion des communications

Comme nous l'avons vu au chapitre 3, nous utilisons des boites aux lettres pour modéliser les communications entre tâches. L'utilisation de tâches conditionnelles entraîne des contraintes particulières pour l'utilisation des communications afin de permettre le repect des contraintes temporelles. A l'évidence l'utilisation des primitives temps réel de communication à l'extérieur des blocs conditionnels d'une tâche conditionnelle revient à une utilisation normale, c'est à dire identique au cas de tâches non conditionnelles.

Néanmoins, comme il n'est pas possible de prévoir lequel des comportements d'exécution d'une tâche conditionnelle va être effectivement exécuté, tenter de faire communiquer une tâche avec l'un de ses comportements met le respect des contraintes temporelles en danger. En effet, dans le cas où un comportement d'exécution d'une tâche conditionnelle envoie un message :

- soit, le message est à destination d'une tâche non conditionnelle et dans ce cas cette dernière peut être bloquée en attente si l'envoi ne se fait pas, c'est à dire si le comportement d'exécution déclenchant le message n'est pas effectivement exécuté.

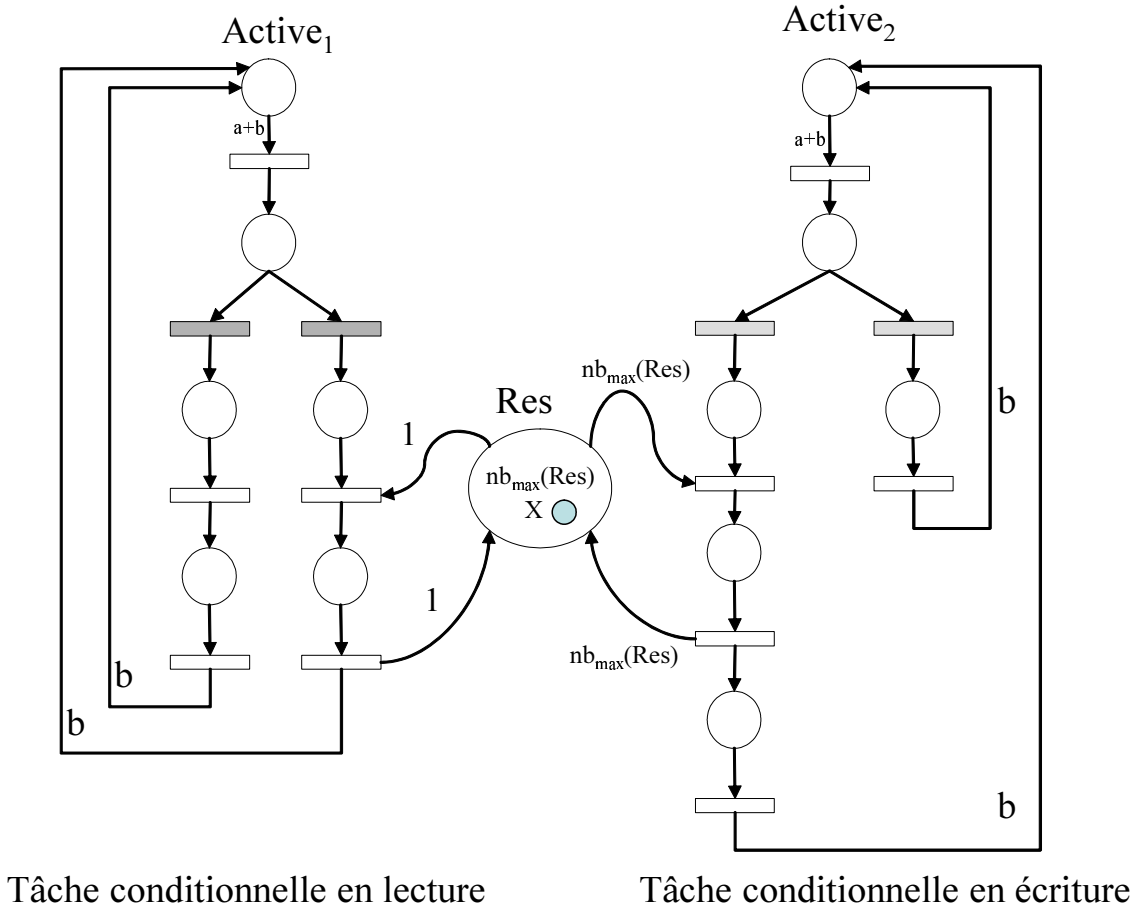

Fig. 5.6: Partage d'une ressource en mode Lecture/Ecriture entre deux tâches conditionnelles.

– soit, il est à destination d'un autre comportement d'exécution d'une autre tâche, et dans ce cas, cette dernière peut rester bloquée pour la même raison que précédemment, mais il est également possible que la boite aux lettre engrange des messages qui ne seront jamais reçus (si le comportement récepteur ne s'exécute pas).

Par conséquent, l'utilisation des communications entre tâches est restreinte par les contraintes suivantes :

– une fréquence identique pour l'émission et la réception de messages,
– l'attente de messages ne peut avoir lieu à l'intérieur d'une section critique,
– pas de réception dans un bloc conditionnel d'une tâche conditionnelle,
– pas d'émission dans un bloc conditionnel d'une tâche conditionnelle vers une tâche périodique.

Toutefois, nous pouvons constater que l'utilisation d'envoi de message à l'intérieur des blocs conditionnels engendre l'émission d'un message de façon aléatoire puisque cela dépend du choix résultant du test conditionnel lors de l'exécution de la tâche. Il semble alors naturel d'essayer de prendre en compte ces messages aléatoires comme déclencheur de tâches sporadiques.

5.2 Modélisation des tâches sporadiques

Nous avons présenté au chapitre 1 la notion de tâche sporadique. Pour rappel, ces dernières comportent trois paramètres : la pire durée d'exécution C_{si}, un délai critique D_{si} et une pseudo période T_{si} correspondant à la durée minimale entre deux activations successives. Nous avons également adopté l'hypothèse que $D_{si} = T_{si}$. Nous présentons par la suite deux modélisations par RdP de ce type de tâche, lorsqu'elles sont déclenchées par une tâche conditionnelle.

5.2.1 Tâches sporadiques déclenchées

L'utilisation des boites aux lettres pour la communication entre tâches pouvant être conditionnelles a mis en évidence la possibilité d'envoi de message uniquement lorsqu'un comportement spécifique d'exécution d'une tâche conditionnelle est exécuté. Nous pouvons utiliser ce principe pour activer une tâche sporadique. La construction par RdP d'une telle tâche nécessite toutefois quelques précautions :

- de par notre modélisation, nous ne pouvons pas gérer convenablement la réentrance des tâches. Nous ne pouvons donc accepter qu'une tâche conditionnelle active une tâche sporadique alors que celle-ci possède déjà une instance en exécution. De fait, cette restriction exprime de façon détournée l'existence d'une pseudo période.
- nous devons pouvoir tenir compte du délai critique des tâches sporadiques.

Prise en compte de la non réentrance

Pour nous assurer de la non réentrance, nous devons signaler au système que l'exécution précédente de la tâche sporadique est terminée. Ceci peut être fait en envoyant par exemple un message dans une boite aux lettres à destination de la tâche déclenchante. Cette dernière est alors obligée d'attendre la présence d'une message de terminaison avant d'envoyer à son tour un message déclenchant à nouveau la tâche sporadique. Toutefois, si plusieurs tâches conditionnelles souhaitent activer une même tâche sporadique, cette méthode engendre l'utilisation de deux boites aux lettres par tâche déclenchante (une pour l'activation et la deuxième pour la terminaison de la tâche sporadique). De plus , il n'existe aucune corrélation entre les boites aux lettres ce qui ne permet pas de garantir la non réentrance de la tâche sporadique. C'est pourquoi, nous intégrons directement les deux places formant initialement les boites aux lettres d'activation et de terminaison dans la structure même de la tâche sporadique. La modélisation d'une telle tâche devient alors un bloc de durée où chaque transition est en concurrence pour le jeton processeur, précédé d'une place recevant le jeton symbolisant l'activation de la tâche (similaire aux places $Activ_i$ sans l'horlogerie locale pour les tâches périodiques) et terminé par une place recevant un jeton uniquement lorsque la tâche sporadique est terminée. La figure 5.7 illustre une modélisation par RdP d'une tâche sporadique déclenché par une tâche conditionnelle. Nous pouvons noter que cette modélisation n'occasionne pas de construction de système d'horlogerie et donc aucun ensemble terminal.

Cette première méthode de modélisation à l'avantage d'être très simple et de ne nécessiter ni délai critique ni pseudo-période explicites, d'où l'absence de marquages

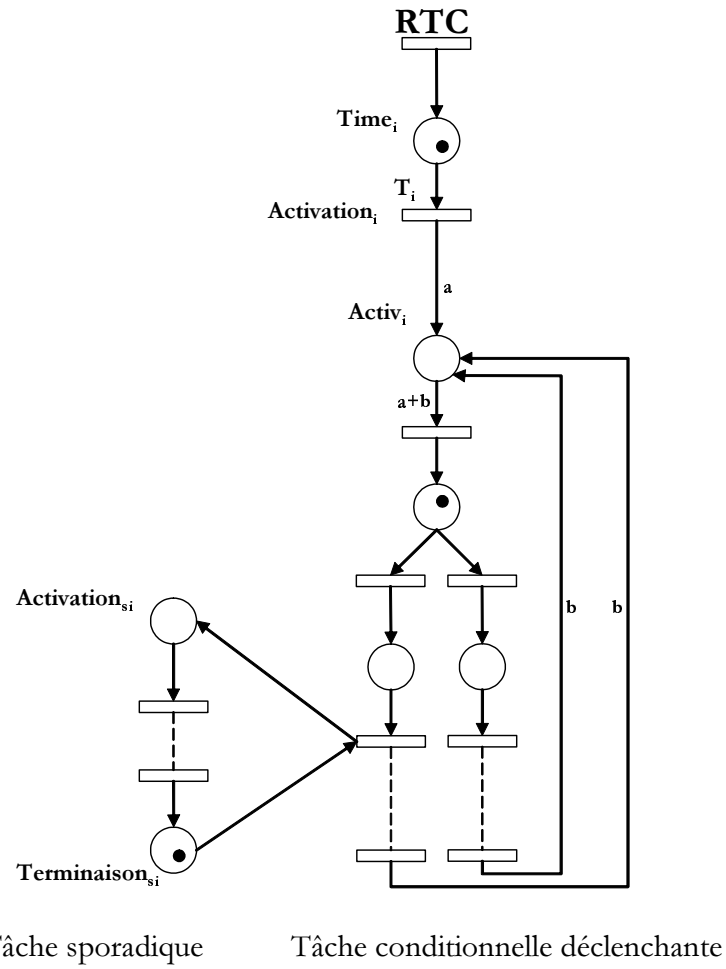

FIG. 5.7: Méthode 1 : Modélisation d'une tâche sporadique déclenchée par une tâche conditionnelle.

terminaux. En contrepartie, l'absence de délai critique peut poser problème pour les tâches conditionnelles déclenchantes. En effet, du fait de la place $terminaison_{si}$ de la tâche sporadique, l'exécution du comportement déclenchant peut être retardée et aboutir à un dépassement d'échéance. En fait, nous ne garantissons que la non réentrance et le délai critique (implicite) dépend du comportement de l'application : si nous choisissons deux fois de suite le comportement déclenchant, il sera réduit, mais si nous restons longtemps sans re-déclencher la tâche sporadique, il peut devenir grand. Toutefois, l'utilisation de contrainte a priori lors de la construction du graphe des marquages peut permettre un meilleur contrôle comme nous le verrons ultérieurement. Malgré cela, nous proposons une deuxième modélisation des tâches sporadiques intégrant explicitement le délai critique dans la modélisation.

Prise en compte du délai critique

Pour tenir compte des délai critiques des tâches périodiques, nous utilisons des marquages terminaux pour vérifier à tout instant le temps écoulé en utilisant le

marquages des les places $Time_i$. qu'au moment où le jeton b est produit, le nombre de jetons de la place $Time_i$ n'est pas supérieur au délai critique de la tâche. En partant de ce principe, nous adjoignons à la tâche sporadique (constituée d'une succession de transitions formant la durée C_{si}) un système d'horlogerie local permettant également de comptabiliser le temps entre une activation et la terminaison associée. Or, à l'inverse des tâches périodiques, ce n'est pas le système d'horlogerie qui doit activer la tâche sporadique mais une transition d'une tâche conditionnelle. Une première conséquence est la nécessité de consommer tous les jetons issus de la tâche RTC (horloge globale) lorsque la tâche sporadique n'est pas activée. Pour cela nous utilisons une transition $Idle_{si}$ qui consomme chaque jeton de la place $Time_{si}$ (horlogerie locale de la tâche sporadique) tout en consommant et produisant le jeton d'une place nommée Is_Idle_{si}. Dès lors, tant qu'il y a un jeton dans la place Is_Idle_{si}, la tâche sporadique est considérée comme inactive. Ceci implique alors que l'activation de cette tâche s'effectue en consommant l'unique jeton de Is_Idle_{si}. Si ce jeton est consommé, la transition $Idle_{si}$ ne peut plus consommer les jetons de la place $Time_{si}$, ; celle ci comptabilise alors le temps. Pour réellement activer la tâche sporadique, il faut donc consommer le jeton de la place Is_Idle_{si} et produire un jeton a dans la place $activ_{si}$ de la tâche sporadique. Lorsque la tâche sporadique termine son exécution, celle ci doit permettre à la place Is_Idle_{si} de retrouver un jeton pour continuer de vider la place $Timesi$. Pour cela, nous rajoutons un arc reliant la dernière transition de la tâche sporadique à la place Is_Idle_{si} pour produire un jeton. Malheureusement, la présence de ce jeton dans Is_Idle_{si} ne va permettre que de consommer chaque nouveau jeton venant de RTC dans $Time_{si}$. Or, il nous faut également éliminer les jetons qui se sont accumulés dans cette place pour pouvoir comptabiliser le temps à la prochaine activation. C'est pourquoi nous étendons les RdP aux RdP synchronisés uniquement pour la transition $Idle_{si}$ qui devient dépendant de l'évènement (e) qui pour rappel est l'évènement toujours occurrent. Ainsi, dès que la jeton revient dans la place Is_Idle_{si}, la transition $Idle_{si}$ est franchie en permanence tant qu'il y a des jetons à consommer dans Is_Idle_{si} et surtout dans $Time_{si}$. La figure 5.8 illustre la modélisation de tâche sporadique déclenchée grâce aux RdP synchronisés.

Le fait de pouvoir comptabiliser le temps entre chaque activation et terminaison des instances de tâche sporadique nous permet de rajouter un marquage terminal qui tient compte des délai critiques pour chaque tâche sporadique.

$$M(Time_{si}) \leq D_{si}$$

Nous pouvons noter que dans cette modélisation, la réentrance d'une tâche sporadique est impossible. En effet, tout comme dans la première méthode la place Is_Idle_{si} joue le rôle de la place terminaison, et oblige ainsi à n'activer la tâche sporadique que si elle est dans l'état inactif.

5.2.2 Implications sur la tâche oisive

Quelque soit la méthode de modélisation des tâches sporadiques utilisée, le temps processeur alloué à ces tâches n'a pas été comptabilisé pour le calcul de la durée de la tâche

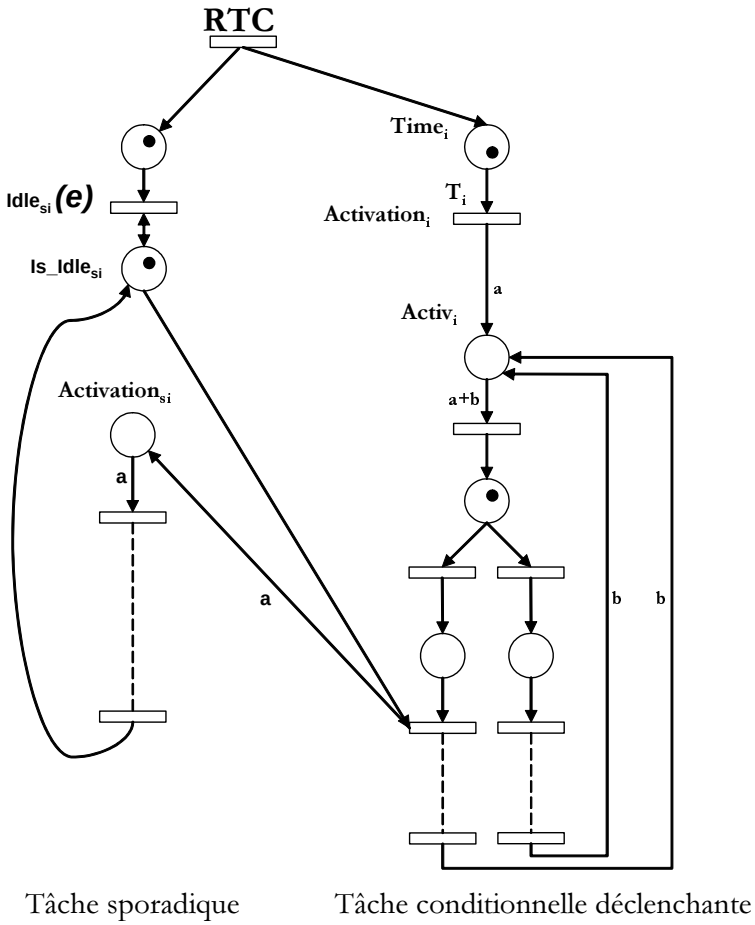

FIG. 5.8: Méthode 2 : Modélisation d'une tâche sporadique déclenchée par une tâche conditionnelle avec prise en compte de son délai critique.

oisive. Ceci signifie que chaque transition des tâches sporadiques représente en l'état une unité de temps faisant passer la charge processeur à plus de 100%. Les tâches sporadiques doivent par conséquent utiliser pour être exécutées, non seulement la ressource processeur, mais également autant de jetons de la tâche oisive qu'il y a de transitions pour modéliser la durée de la tâche sporadique. Nous devons donc rajouter un arc reliant la première transition de la tâche sporadique à la place $Activ_0$ en le pondérant par C_{si}, la durée de la tâche sporadique. Ainsi, nous pouvons assurer par avance qu'il y aura pas de surcharge processeur.

Nous aurions pu également relier chaque transition de la tâche sporadique correspondant à sa durée d'exécution, à la place $Activ_0$ par un arc de poids unitaire. Ceci aurait permis de profiter des éventuels temps creux rajoutés à la tâche oisive par l'exécution d'un comportement de durée minimale d'une tâche conditionnelle. Mais, si cette solution augmente la puissance d'ordonnançabillité, elle engendre également de nombreux scénarios d'ordonnancement aboutissant à des fautes temporelles. En effet, cette solution anticipe sur les éventuels rajouts de temps creux dans la tâche oisive. La figure 5.9 illustre l'utilisation des jetons de la tâche oisive par les tâches sporadiques quelque soit la méthode de modélisation.

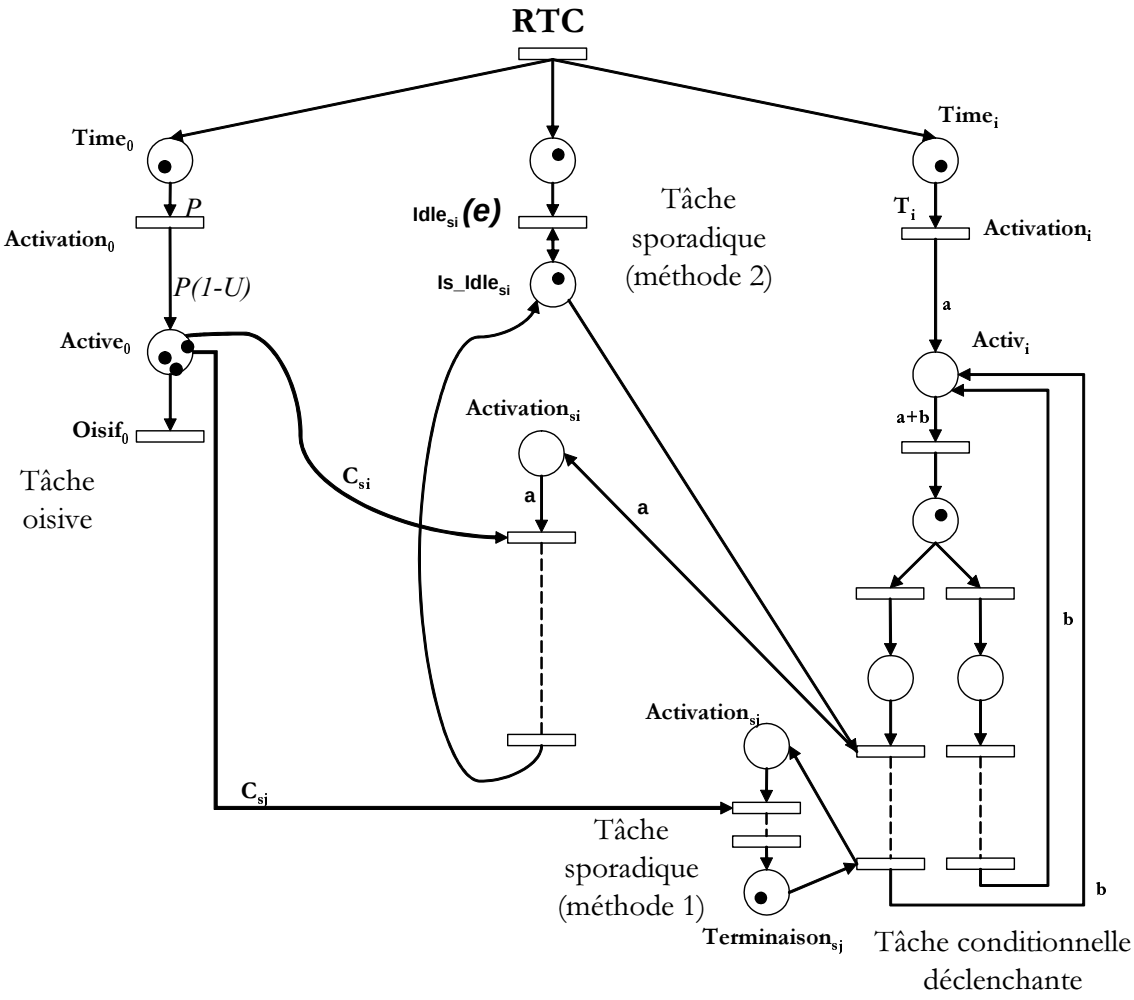

Fig. 5.9: Prise en compte des temps creux pour la gestion des tâches sporadiques. Dans un souci de clarté, nous ne représentons pas ici le lien entre la tâche conditionnelle et la tâche oisive.

5.3 Étude du graphe des marquages

Nous nous intéressons ici à l'exploitation du graphe des marquages induit par la modélisation par réseau de Petri.

Comme nous considérons uniquement les ordonnancements valides, nous cherchons à obtenir les mots du langage engendré par le RdP pour lesquels chaque état respecte les contraintes terminales permettant de s'assurer de la validité temporelle (en accord avec l'étiquetage des transitions vu au chapitre 3).

Par la suite, nous appellerons *Graphe d'Accessibilité*(noté *GA*), le graphe des marquages réduit aux seuls franchissements des transitions issues de la modélisation des durées des tâches périodiques, sporadiques et oisive.

Au même titre que dans le cas des tâches non conditionnelles, l'étude exhaustive de l'ordonnancement étant NP-difficile, l'extension de la modélisation à des tâches conditionnelles ne peut qu'induire une plus grande explosion combinatoire. Nous devons donc analyser les limites aussi bien spatiales que combinatoires du graphe d'accessibilité dans

le cas des tâches conditionnelles.

Dans un premier temps, nous étudions plus précisément l'implications des tâches conditionnelles dans cette modélisation. Nous regardons par la suite, les difficultés supplémentaire qu'engendre la prise en compte des tâches à départs différés puis des tâches sporadiques. Nous reprenons ici les conditions d'étude vues au chapitre 3. Puis dans un deuxième temps, nous regardons comment extraire les arbres d'ordonnancement valides à partir du graphe d'accessibilité.

5.3.1 Tâches à départs simultanés

5.3.1.1 Taille du GA

L'exploitation directe du réseau de Petri permet d'obtenir un graphe des marquages de profondeur P (la metapériode) dans le cas de tâches à départs simultanées. En effet, le système est dans le même état à une profondeur de t et à $t + k \times P$ (k>0) du fait de la cyclicité des séquences d'ordonnancement. La prise en compte de tâche conditionnelles n'implique par conséquent pas d'augmentation de la longueur du graphe d'accessibilité. Par contre, tenir compte de l'ensemble des comportements des tâches ne peut qu'augmenter la taille du GA. Nous avons vu que dans le cas des tâches non conditionnelles, la taille du graphe était contenu dans un hyperpavé de dimension n+1 où les longueurs des dimensions $1, \ldots, n$ sont respectivement

$$\left\lceil \frac{P \cdot C_i}{P_i} \right\rceil_{i=1\ldots n}$$

et la longueur de la dernière dimension est C_0. Considérons une tache conditionnelle comportant deux instructions conditionnelles. Soit ces instructions sont disposées en série, soit elles sont imbriqués l'une dans l'autre :

- Dans le premier cas, nous pouvons considérer qu'à chaque test conditionnel, soit nous continuons l'exécution dans la même dimension que précédement, soit nous optons pour l'alternative et engendrons alors une autre dimension puisqu'il faut pouvoir dissocier ces deux alternatives. Une fois que le bloc conditionnel est terminé, nous nous retrouvons dans la dimension de départ. Le deuxième test conditionnel pourra à son tour utiliser ces deux dimensions. Nous pouvons donc généraliser en constatant qu'une succession de tests conditionnels n'engendre qu'une seule dimension supplémentaire dans la taille de l'hyperpavé.
- Dans le second cas, en suivant le même raisonnement, nous nous apercevons qu'une fois les deux dimensions créées, c'est à dire le premier test conditionnel effectué, il reste encore dans l'une des dimension, un deuxième test conditionnel. La construction de ce deuxième test engendre donc pour dissocier les alternatives une troisième dimension. Nous pouvons alors généraliser sur le fait que des test conditionnels imbriqué engendrent une dimension supplémentaire par test conditionnel.

La propriété suivante définit de façon formelle, la taille du GA.

Propriété 9 ***Taille du graphe d'accessibilité en présence de tâches conditionnelles***

Considérons n tâches définies par $< r_i, \zeta_i, D_i, T_i >$ *pour* $1 \leq i \leq n$*, ne comportant pas de primitives temps réel : l'ensemble des graphes d'ordonnancement est entièrement contenu dans un hyperpavé comportant*

$$\sum_{i=1}^{n}(\Phi(\zeta_i)) + 1 \text{ cotés,}$$

avec

$$\Phi(\zeta_i) = \begin{cases} 0 \text{ si } \#(\zeta_i) = 1 \\ \Phi(IfThen\ B_1\ Else\ B_2\ EndIf\ B_3) = max\,(B_3, 1 + max\,(\Phi(B_1), \Phi(B_2))) \end{cases}$$

 # représentant le nombre d'éléments de ζ_i
$\Phi(\zeta_i)$ *représente le nombre de tests conditionnels imbriqués parmi les comportements de la tâche i.*

Les longueurs de ses côtés sont toutefois impossibles à donner. En effet, pour chaque test conditionnel, l'une des alternatives emprunte la dimension déjà existante. Or, comme ce choix est arbitraire, il est nécessaire de définir précisément l'attribution de chaque comportement d'exécution à la dimension qu'elle va emprunter.

5.3.1.2 Construction du GA

Le graphe des marquages comporte des états sans successeurs, c'est à dire des branches ne permettant pas d'arriver à la hauteur P en raison d'une erreur d'ordonnancement (marquages terminaux non respectés). Il est donc nécessaire d'épurer le graphe pour obtenir au final un *graphe d'accessibilité* qui contient toutes les séquences d'ordonnancement valides pour chaque chemin d'exécution des tâches. C'est la méthode utilisé dans [Gro99]. Si nous optons pour cette approche telle quelle nous nous cantonnons au contexte d'ordonnancement local. En effet, la dualité des instructions conditionnelles n'est pas respectée à ce stade.

Nous nous intéressons à l'ordonnancement global pour extraire un arbre d'ordonnancement certifiant l'existence d'un chemin d'exécution quelque soit le résultat des instructions conditionnelles. Nous devons donc une nouvelle fois réduire le graphe d'accessibilité en suivant la règle de la propriété 10 pour tenir compte du caractère global de l'ordonnancement.

Propriété 10 ***Contrainte de construction du GA pour l'ordonnancement global***

Soit le marquage M et les transitions t^+ *et* t^- *représentant une instruction conditionnelle issues de M, alors les marquages provenant de* t^+ *sont valides si et seulement si ceux issus de* t^- *le sont également.*

A chaque instant où un choix est fait sur le chemin d'exécution à emprunter, le choix de l'autre chemin correspondant à l'alternative doit être possible. Nous obtenons alors un **graphe d'accessibilité globale**.

Comme cela a déjà été dit au chapitre 3, le GA possède un unique état initial M_0 qui dans le cas de tâches à départs simultanés est un état d'accueil, c'est à dire un état accessible à partir de tout marquage accessible. Á partir de cet état initial, la construction du graphe, facilitée par le fait que chaque place est bornée et que la profondeur du GA est bornée et connue, peut se faire en suivant l'une ou l'autre des deux stratégies décrites ci-après :

- Construction en largeur d'abord ; cette construction est utilisée si nous souhaitons construire l'ensemble de toutes les arborescences valides, afin de choisir la meilleure d'entre elles pour un critère qualitatif donné (par exemple, pour optimiser le temps de réponse moyen d'un ensemble de tâches). Á partir de l'état M_0 de hauteur 0, nous construisons autant de fils à M_0 qu'il y a de transitions franchissables en accord avec les ensembles terminaux. Ces fils sont ainsi à profondeur 1 et nous opérons de la même façon pour chacun d'entres eux. La construction finale du GA s'obtient hauteur après hauteur jusqu'à la profondeur P qui ne comporte qu'un unique état M_f tel que $M_f = M_0$. Cette construction entraîne de nombreux retours arrière (backtracking) puisqu'un marquage inséré à la hauteur h peut ne plus avoir de fils à la hauteur h+k. Ce marquage est considéré comme stérile et doit donc être supprimé du GA. Pour éviter un grand nombre de retours arrière nous avons recours à une table de production basée sur la latence des tâches. Le principal inconvénient de cette méthode est la nécessité de construire l'ensemble du GA avant de pouvoir exhiber une solution ou tout simplement conclure au problème de l'ordonnançabilité.
- Construction en profondeur d'abord ; cette technique est utilisée préférentiellement lorsque l'objectif recherché est l'obtention d'une arborescence valide. Ainsi le graphe entier n'est pas construit inutilement comme ce serait le cas avec une construction en largeur. Cette construction permet à chaque étape de construire un seul noeud du GA d'une hauteur immédiatement supérieure. Ainsi, à partir de M_0 nous construisons un seul fils de hauteur 1 en accord avec les ensembles terminaux. L'étape suivante consiste alors à chercher un fils de hauteur 2 *etc.*... L'avantage principal de cette méthode est qu'il suffit d'arriver à une hauteur P pour conclure à l'ordonnançabilité de l'application. Toutefois, dès lors que nous chercherons à exhiber une solution correspondant à certains critères, la construction complète du GA sera nécessaire. Dans ce cas, la construction en profondeur occasionne un nombre beaucoup plus conséquent de retours arrière que la méthode en largeur.

Dans ces deux cas, l'arbre construit permet l'analyse de l'ordonnançabilité locale. Pour aborder le problème de l'ordonnançabilité globale, il faut en plus intégrer la propriété 10. Dans le cas d'une construction en profondeur, l'intégration des alternatives des tests conditionnels ne pose guère de problème. Nous mémorisons, au fur et à mesure de la construction, les marquages par lesquels nous avons construit un fils en utilisant une alternative d'une conditionnelle. Une fois la profondeur P atteinte, nous remontons afin de traiter, pour ces marquages, l'autre alternative. Si, à partir de chacun des marquages, en prenant l'autre alternative, nous pouvons atteindre la profondeur P, alors nous aurons construit un arbre d'ordonnancement et l'application est globalement ordonnan-

çable. Toutefois, cette méthode augmente considérablement les cas de retour arrière : la construction d'un même marquage stérile peut apparaître plusieurs fois après chaque suppression. C'est pourquoi, si nous souhaitons construire l'ensemble du GA nous devons adopter la méthode en largeur.

La méthode en largeur d'abord comme nous l'avons vu permet une construction globale du GA sans occasionner trop de retours arrière. La prise en compte des tâches conditionnelles ne peut être appliquée durant la construction. En effet, il nous est impossible de savoir si les alternatives d'une construction conditionnelle de hauteur h seront toutes validées à la hauteur P. Nous devons par conséquent construire préalablement un GA local ne tenant pas compte de la dualité des instructions conditionnelles. Une fois ce GA construit, nous devons ensuite épurer ce graphe en vérifiant pour chaque noeud que la propriété 10 est bien vérifiée. L'algorithme de cette épuration est le suivant : nous supposons que chaque noeud M est caractérisé par :

- Un marquage correspondant à un état du système,
- un objet *cond* composé d'une variable booléenne test qui indique s'il existe au moins un arc issu du noeud M étiqueté par une condition, une liste de doubles liens (C^+, C^-) correspondant aux alternatives de chaque test et une fonction $nbCond$ permettant de connaître le nombre de doubles liens,
- une liste d'arcs $LienPre$, qui stocke tous les arcs qui pointent sur le noeud M,
- une liste d'arcs $LienFils$, qui stocke tous les arcs issus de M (qu'ils soient conditionnels ou non),
- une fonction $nbfils(M)$ permettant de connaître le nombre d'arcs de $LienFils$.

```
PROCEDURE EpurationGraphe(Graphe)
Booleen GrapheModifié=TRUE ;
DEBUT
 Tant que (GrapheModifié) faire
  -- le graphe a été modifié, il peut y avoir des noeuds stériles
  GrapheModifié := FALSE ;
  Pour tout Marquage Mi faire
   Si (Mi.cond.test) alors
    -- il y a des liens tests issus de Mi
    Pour tout lien conditionnel j issu de Mi faire
     Si (Mi.cond.C+[j]=NIL) alors
      -- l'alternative C+ n'existe pas
      SupprimerLien Mi.cond.C-[j]
      DétruireFils (Mi.Cond.C-[j])
      GrapheModifié=TRUE ;
     FinSi
     Si (Mi.cond.C-[j]=NIL) alors
      -- l'alternative C- n'existe pas
      SupprimerLien Mi.cond.C+[j]
      DétruireFils (Mi.Cond.C+[j])
      GrapheModifié=TRUE ;
     Finsi
```

```
     FinPour
    Si (Mi.Cond.nbCond()=0) alors
     -- Il n'y a plus de liens tests issus de Mi
     Mi.cond.test=FALSE
    FinSi
   FinSi
   Si (NbFils(Mi)=0) alors
    -- Mi est stérile
    Pour tout lien Père j issu de Mi faire
     SupprimerLien Mi.LienPère[j]
     DétruirePère( Mi.lienPére[j])
     GrapheModifié :=TRUE ;
    FinPour
   FinSi
  FinPour
 FinTantque
FIN

-- Détruit l'arborescence de sommet M
PROCEDURE DetruireFils( Marquage M)
DEBUT
 Si (NbPère(M)=0) alors
  -- Il n'y a aucun lien
  -- pointant sur M
  Pour tout lienfils L faire
   SupprimeLien L
   DetruireFils (M.L)
  Fin Pour
  SupprimeMarquage M
 FinSi
FIN

-- Détruit un marquage M stérile et
-- ses pères devenus également stériles
-- par cette suppression
PROCEDURE DetruirePère(Marquage M)
DEBUT
 Si (NbFils(M)=0) alors
  -- M est stérile
  Pour tout lienPèrej issu de M faire
   SupprimerLien M.LienPère[j]
   DetruirePère (M.LienPère[j])
  FinPour
  SupprimerMarquage M
 FinSi
```

FIN

Nous donnons dans la figure 5.10, un exemple de la construction d'un graphe d'accessibilité global pour une application temps réel comportant deux tâches. Une tâche conditionnelle de période la métapériode et une tâche à durée du pire cas s'exécutant durant deux instances pendant la métapériode. Nous pouvons constater sur cet exemple simple l'augmentation de la taille du GA qu'engendre une tâche conditionnelle.

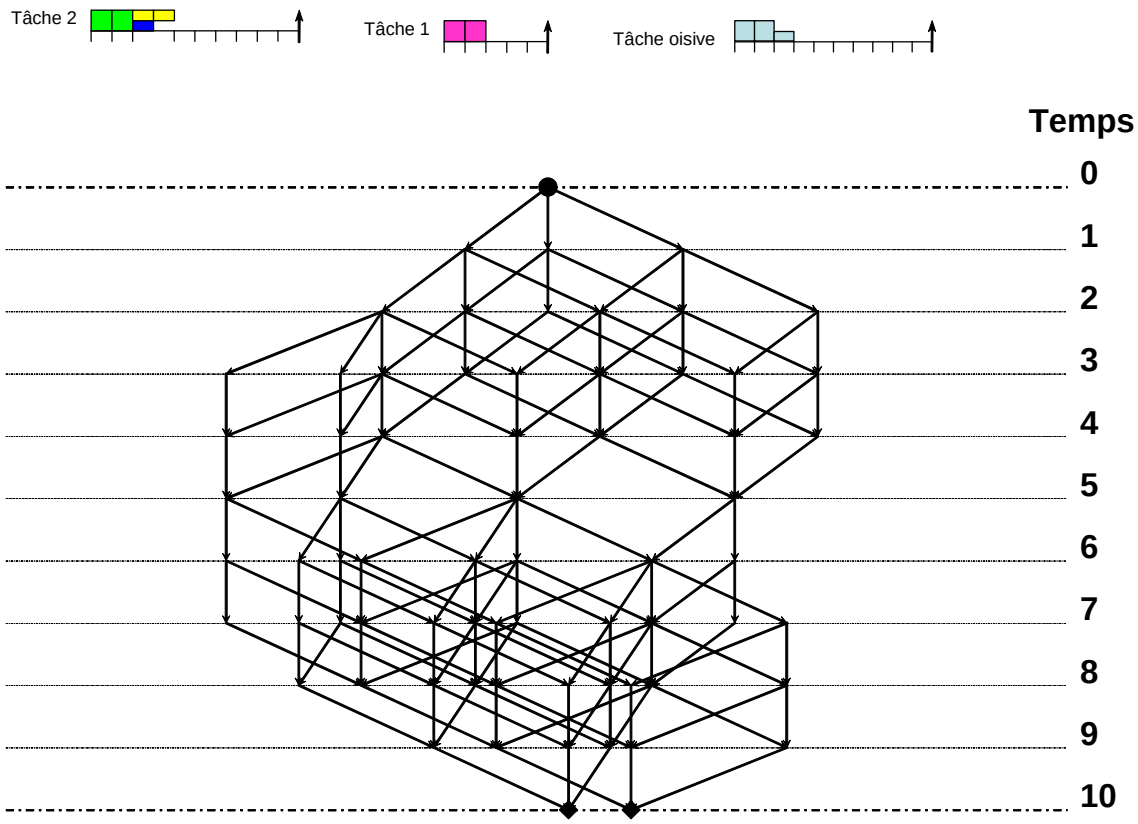

Fig. 5.10: Exemple de la construction d'un graphe d'accessibilité global d'une application temps réel composée de deux tâches : une tâche conditionnelle et une tâche périodique à durée d'exécution du pire cas. La troisième tâche est la tâche oisive dont la durée dépend des alternatives de la tâche conditionnelle.

5.3.2 Tâches à départs différés

Nous venons de voir la construction du GA global dans le cas des tâches à départs simultanés. Nous avons vu que la prise en compte des tâches conditionnelles augmente la taille du GA surtout si les instructions conditionnelles sont imbriquées, ce qui accroît fortement la complexité spatiale de notre approche.

Si nous voulons prendre en compte les tâches à départs différés nous devons, compte tenu des résultats sur la cyclicité, augmenter la profondeur du GA. Cette augmentation

du nombre de noeuds du graphe accroît alors fortement la complexité de construction de celui-ci. De plus, contrairement aux tâches non conditionnelles pour lesquelles le marquage M_f survient au temps $t_c + 1 + P$, dans notre cas, nous pouvons mettre en évidence plusieurs marquages M_f, correspondant à différents comportements d'exécution des tâches conditionnelles.

Nous savons d'après la propriété 8 que la date $t_c + 1 + P$ en considérant t_c comme la date du dernier temps creux acyclique pour les comportements de plus longue durée pour chaque tâche est maximale. Il s'en suit que l'ensemble des marquages M_f survient à des hauteurs h_i inférieures et correspondant à des états du GA déjà construits de hauteur $h_i - P$. Les critères de terminaison deviennent alors complexes : soit nous construisons le GA jusqu'à la hauteur $t_c + 1 + P$ sans pour autant avoir un unique M_f, soit nous vérifions dès la profondeur P qu'aucun marquage construit n'est en réalité un marquage M_f pour les comportements d'exécution effectivement exécutés pour arriver à ce marquage.

5.3.3 Tâches sporadiques déclenchées

Nous nous plaçons ici dans le cas de tâches périodiques à départs simultanés. Nous avons vu précédemment comment modéliser les tâches sporadiques déclenchées. La prise en compte de ces dernières ne pose pas de problème majeur pour la construction du GA global. En effet, les tâches sporadiques utilisent les temps creux de la tâche oisive pour s'exécuter, ainsi s'il n'y a pas suffisamment de temps creux, les tâches sporadiques sont bloquées et engendrent des marquages non compatibles avec les ensembles terminaux.

Nous pouvons, avant toute construction, vérifier que les tâches sporadiques n'engendrent pas de surcharge fatale au bon fonctionnement de l'application. Pour cela, nous calculons la pire charge que le système doit supporter. Considérons un système de n tâches périodiques τ_i avec $1 \leq i \leq n$ et p tâches sporadiques τ_{s_j} avec $1 \leq j \leq p$. Nous notons également $\tau_{i,j}$ le j^{ieme} comportement d'exécution de la tâche τ_i, $Nb(i,j)$ et $Vect_{C_s}$ les vecteurs de dimension p à valeur dans $\mathbb{N}$ respectivement d'indice des tâches sporadiques déclenchées par $\tau_{i,j}$ et de durée d'exécution C_{s_j}. (Nous considérons qu'un comportement d'exécution d'une tâche conditionnelle ne peut déclencher plus d'une fois chaque tâche sporadique par instance)

$$U_{sporadique} = \sum_{k=1}^{n} \frac{MAX_{\forall l}(C_{k,l} + Nb(i,j)^t \times Vect_{C_s})}{T_k}$$

Si $U_{sporadique} > 1$, alors le système ne peut pas supporter les tâches sporadiques et continuer à garantir le respect des contraintes temporelles. Dans le cas où $U_{sporadique} \leq 1$, il nous faut alors réaliser l'étude et donc construire la graphe d'accessibilité.

Dans le cas où les tâches sporadiques ont été modélisées avec des contraintes temporelles, cette définition de $U_{sporadique}$ ne permet toutefois pas de conclure, puisque le système est sur-contraint de par la présence d'une échéance.

Par ailleurs, un autre problème se pose. Comme nous pouvons le constater sur la figure 5.11, à la hauteur P (metapériode = 10 dans l'exemple) la tâche sporadique ne s'est pas encore terminée.

En effet, en considérant qu'une tâche conditionnelle déclenche une tâche sporadique lors de sa dernière instruction, à la hauteur P qui correspond à la fin du cycle du système, il reste encore cette tâche à exécuter. Or, il n'est pas possible de boucler sur l'état M_0

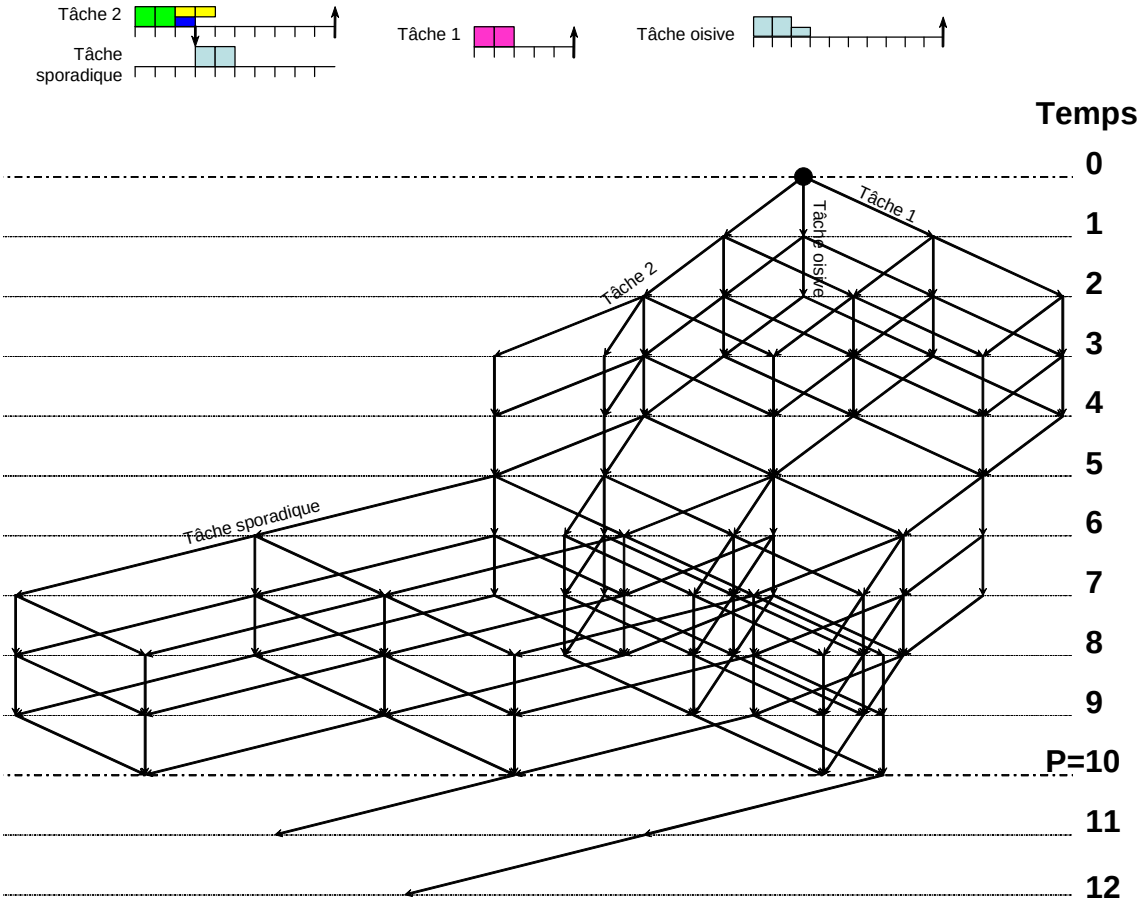

Fig. 5.11: Exemple de la construction d'un graphe d'accessibilité global d'une application temps réel composée de trois tâches : une tâche conditionnelle déclenchant une tâche sporadique et une tâche périodique à durée d'exécution du pire cas. Nous pouvons constater que l'exécution de la tâche sporadique n'est pas terminée lorsque les tâches périodique atteignent le marquage final de hauteur la métapériode P=10.

puisqu'à cet état il n'y a au départ aucune tâche sporadique déclenchée. Intuitivement, nous devons construire le GA jusqu'à une profondeur L suffisante pour s'assurer que toutes les tâches sporadiques soient terminées. Une telle profondeur peut être évaluée sachant que ce sont les tâches conditionnelles qui les déclenchent. Ainsi nous pouvons choisir L tel que

$$L = P + max_{i \in Cond(\tau)}(T_i)$$

avec $Cond(\tau)$ l'ensemble des indices des tâches conditionnelles déclenchantes.

De cette façon, l'ensemble des noeuds de hauteur L existe déjà à la hauteur L-P, tâches sporadiques comprises puisqu'une instance de chaque tâche déclenchante a déjà été exécutée à cette date. Si cette méthode est applicable pour la construction en largeur, elle occasionne malheureusement deux effets de bord majeurs :

- Une forte augmentation de la taille du GA et une multitude de marquages M_f.
- Un arbre d'ordonnancement n'est plus de longueur borné par $[0, L]$.

Quand nous considérons un arbre d'ordonnancement A, un noeud M_f de profondeur

$h_f > P$ a, dans le GA, un ancêtre identique à profondeur $h_f - P$ mais le problème est que cet ancêtre n'appartient pas nécessairement à l'arbre d'ordonnancement A. L'exemple 5.11 illustre ce phénomène. En effet, si nous considérons l'arbre d'ordonnancement permettant d'aboutir à la fin de l'exécution de la tâche sporadique au temps 12 (la métapériode (10) est dépassé de 2 unité de temps), l'état du système est alors identique à un état au temps 2 correspondant à deux exécutions de la tâche oisive (pour compenser l'exécution de la tâche sporadique pendant 2 unités de temps après la métapériode). Or, cet état au temps 2 ne fait pas parti de l'arbre d'ordonnancement initial permettant de terminer l'exécution de la tâche sporadique au temps 12. Il n'est donc pas évident d'exhiber un arbre d'ordonnancement valide incluant des tâches sporadiques.

Une autre méthode consiste alors à utiliser une construction en profondeur jusqu'à la hauteur P et à vérifier que les tâches sporadiques peuvent remplacer les temps creux par les durées des tâches depuis l'état initial M_0 tout en respectant les délais critiques des tâches sporadiques. Cette méthode permet plus facilement d'exhiber une solution mais ne permet pas d'appliquer des critères d'optimalité sur l'arbre d'ordonnancement construit.

5.3.4 Contraintes absolues

Nous avons vu que la prise en compte des tâches conditionnelles engendre une nette augmentation de la taille du GA. Malgré la présence de primitives temps réel qui diminue fortement le nombre d'arbres d'ordonnancement dans le graphe d'accessibilité global, il est nécessaire d'éviter sa construction dans son ensemble. C'est pourquoi, nous adoptons des techniques d'optimisation par contraintes [Gro99] permettant de réduire la construction du graphe d'accessibilité globale pour la recherche d'un arbre d'ordonnancement.

Une première méthode consiste à reprendre le critère de contraintes de successeurs vu au chapitre 3. Cette contrainte ne remet pas en cause l'ordonnançabilité de l'application et permet de supprimer du GA global, l'ensemble des marquages issus d'une préemption inutile due aux entrelacement entre tâches. Les tâche conditionnelles tout comme les tâches sporadiques ne provoquent pas de perte de la puissance d'ordonnançabilité lorsque nous utilisons cette contraintes durant la construction du GA. Plus précisément, nous ne perdons pas de critère d'optimalité. La figure 5.12 illustre l'application de cette contrainte en l'appliquant à l'application de la figure 5.11.

Il peut être également intéressant de borner certaines caractéristiques temporelles des tâches pendant la construction dans le but de minimiser la taille du GA résultant. En effet, si l'arbre d'ordonnancement voulu possède par exemple un temps de réponse borné pour une tâche spécifique ou pour l'ensemble des tâches, il est alors souhaitable de prendre en compte ce critère dès la construction de GA global. Pour cela, à chaque construction d'un noeud du GA, nous évaluons le critère préétabli conformément aux travaux de [Gro99], puis nous déclarons le noeud valide ou stérile suivant qu'il vérifie ou non la contrainte absolue prédéfinie. Comme cela a déjà été vu au chapitre 3, ces contraintes absolues peuvent porter sur les temps de réponse, la laxité ou encore le taux de réaction des tâches. Nous pouvons appliquer ces contraintes sans le moindre changement puisqu'ils sont appliqués pendant la construction c'est à dire en une phase descendante vis à vis du GA. Par conséquent, les instructions conditionnelles rencontrées pendant la construction sont traitées comme s'il s'agissait de l'ordonnançabilité locale, c'est à dire

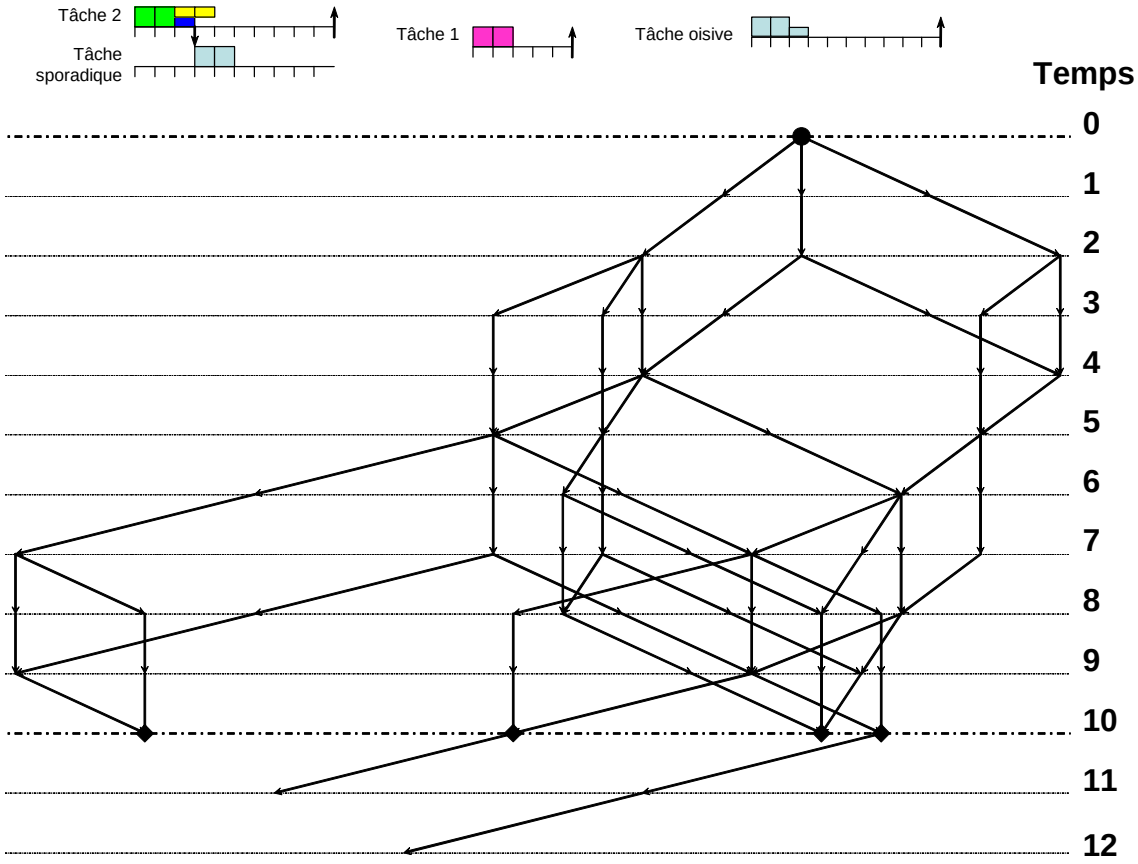

Fig. 5.12: La construction du graphe d'accessibilité global avec contrainte de successeur permet de réduire considérablement sa taille du GA de l'application de la figure 5.11.

que nous pouvons ici dissocier chaque comportement pour vérifier ces critères de façon indépendante.

5.3.5 Extraction de arbre d'ordonnancement

Nous nous plaçons dans le cas où un GA global a été construit grâce aux méthodes définies précédemment. Nous considérons ici que l'application ne comporte pas de tâches sporadiques et que les tâches périodiques sont à départs simultanés. Nous voulons à présent extraire un arbre d'ordonnancement à partir du GA global qui a pu être restreint en taille par des contraintes absolues. Cette extraction s'avère d'autant plus complexe que l'application comporte de nombreuses tâches conditionnelles et plus particulièrement de nombreux comportements d'exécution possibles par tâche.

Propriété 11 ***Dénombrement des arbres d'ordonnancement du GA***

Considérons n tâches définies par $< r_i, \zeta_i, D_i, T_i >$ *pour* $1 \leq i \leq n$*, ne comportant pas de primitives temps réel :*

– *Le nombre de comportements possibles pour une tâche* τ_i *sur la metapériode P est*

$$(\#\zeta_i)^{\frac{P}{T_i}},$$

– *Le nombre de comportements possibles dans un arbre d'ordonnancement est*

$$\prod_{i=1}^{n}(\#\zeta_i)^{\frac{P}{T_i}},$$

 # représentant le nombre d'éléments de ζ_i*,*

Le graphe d'accessibilité globale est composé de l'ensemble des arbres d'ordonnancement possibles, nous pouvons donc facilement nous rendre compte de l'explosion combinatoire rencontrée lors de l'énumération des arbres d'ordonnancement valides.

Nous donnons un algorithme permettant d'extraire un arbre d'ordonnancement. Cet algorithme s'initialise sur le marquage initial M_0 et permet à chaque étape de faire un choix sur marquage fils permettant de construire l'arbre d'ordonnancement. Autrement dit, à chaque hauteur du GA, il est possible de sélectionner un fils en particulier puisque par définition du GA global, ce marquage permet inévitablement d'aboutir à la hauteur P du GA quelque soit les tests conditionnels effectués. Toutefois, lorsque le marquage choisi correspond à une instruction conditionnelle, il faut également choisir le marquage correspondant à l'autre alternative. Nous utilisons pour l'algorithme d'extraction les notations suivantes :

- une pile *PileCond* qui contient un noeud du GA (Marq) d'où part un test, un arc étiqueté par l'une alternative du test et un noeud de l'arbre d'ordonnançabilité (Place),
- une fonction $Genre(Arc)$ qui indique si un arc est un TEST,
- une fonction $Complementaire(Liste_AC, Arc)$ qui renvoie l'alternative du test Arc parmi les arcs de $Liste\_AC$.

Les arcs sont stockés dans des structures possédant un champ *Label* pour le nom de l'action, c'est à dire le critère de choix sur le fils, un champ *end* pour pointer sur l'extrémité de l'arc.

```
PROCEDURE Genere (Graphe, PileCond, Courant, Arbre)
DEBUT
 Si (Graphe=Graphe Vide) alors
  -- terminaison : arrivée à la hauteur P dans le graphe
  Si (PileCond=Pile Vide) alors
   -- toutes les alternatives ont déjà été prises en compte,
   -- l'arbre d'ordonnancement est complet
   Enregistrer (Arbre)
  Sinon
   -- il reste des alternatives (C-) stockées dans PileCond
   Courant :=PileCond.place
   Graphe :=PileCond.Marq
```

```
   Depiler(PileCond)
   Genere(Graphe, Arbre, PileCond, Courant)
  FinSi
 Sinon
  -- on peut apporter un critère particulier
  -- pour ne choisir qu'un seul arc
  Action :=Choix d'un arc issu de l'état courant du Graphe
  Courant.FilsGauche :=new Noeud(Action.Label, Null, Null)
  Si (Genre(Action)=TEST) alors
   -- le noeud courant est un TEST, on crée un fils droit,
   -- on le stocke dans PileCond
   Action2 :=Complementaire(Graphe Action)
   Courant.FilsDroit :=new Noeud (Action2.Label, Null, Null)
   Empiler(Action2.end, Courant.filsDroit, PileCond)
   Courant.FilsDroit :=Null
  FinSi
  -- on recommence la procédure avec le graphe sous-jacent
  -- pointé par l'arc courant
  Genere(Action.end, Arbre, PileCond,Courant.FilsGauche)
  Courant.filsGauche :=Null
 FinSi
FIN
```

La figure suivante (5.13) illustre sur un exemple l'extraction d'un arbre d'ordonnancement à partir d'un GA global non contraint puis restreint par contrainte de successeur.

5.3.6 Contraintes a posteriori et/ou optimales

Nous pouvons dorénavant extraire un arbre d'ordonnancement à partir du GA global. Nous avons vu qu'il était possible durant l'algorithme d'extraction de choisir quel marquage retenir, en d'autre terme quelle tâche exécuter à chaque hauteur du GA. Il parait alors intéressant de guider ce choix en marquant préalablement chaque marquage du GA par une valeur de pertinence au regard d'un certain critère donné. Nous pouvons ajouter un champ particulier à chaque marquage du GA et utiliser ce dernier comme marqueur pour permettre à l'instruction *Action* de l'algorithme de génération d'arbre d'ordonnancement de faire un choix éclairé.

Parmi les critères envisageables, tous ceux décrits dans [Gro99] sont applicables. Il s'agit de :

- l'exécution au plus tôt,
- la minimisation du temps de réponse Moyen ou Maximal,
- l'optimisation du taux de réaction,
- l'optimisation du retard moyen,
- la minimisation de la gigue.

Toutes ces contraintes ont été vues au chapitre 3, toutefois leur mise en oeuvre impose quelques précisions quand à l'évaluation des marquages représentant une transition d'une

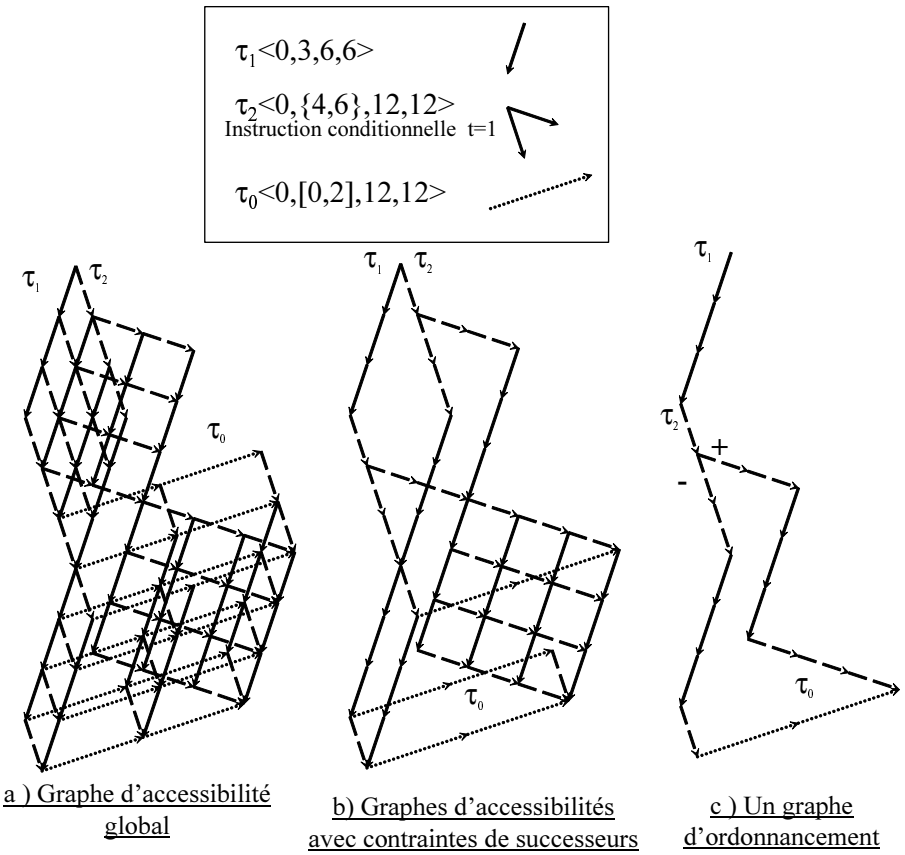

Fig. 5.13: Le graphe d'accessibilité global (a) étant de taille très importante, il est nécessaire de lui appliquer des contraintes lors de sa construction, comme des contraintes de successeur (b), pour pouvoir en extraire un arbre d'ordonnancement (c).

instruction conditionnelle. En effet, l'évaluation des différents critères par une exploration ascendante du GA ne pose aucun problème pour tous les blocs de durée traditionnels qu'ils soient situés à l'extérieur ou à l'intérieur des blocs conditionnels, mais dès lors qu'une transition conditionnelle doit être évaluée, il est nécessaire de tenir compte à la fois de la valeur calculée en provenance du comportement d'exécution correspondant au test positif du test et celle issue du comportement d'exécution du test négatif. Par exemple, en explorant le GA du marquage M_f vers M_0, nous évaluons successivement chaque état en remontant d'une unité de temps dans le GA. Or, en agissant comme cela nous ne savons pas encore quels sont les marquages d'une même hauteur qui feront au final partie du même arbre d'ordonnancement puisque le marquage correspondant à l'instruction conditionnelle se trouve à une hauteur inférieure. Nous devons par conséquent, fusionner les valeurs issues des alternatives d'un test conditionnel dès que nous sommes amenés à évaluer le marquage correspondant à ce test.

Nous pouvons pour cela choisir quatre approches différentes pour définir la valeur du marquage correspondant à l'exécution d'un test conditionnel :

– soit nous prenons systématiquement la valeur minimale (dans le cas d'une minimisation) parmi des deux alternatives. Dans ce cas nous risquons d'avoir un écart de

valeur important entre les deux branches issues du test conditionnel mais cela permet d'avoir un nombre maximum de candidats lors de la phase d'extraction. C'est une vison optimiste de la contrainte.
- soit nous prenons systématiquement la valeur maximale (dans le cas d'une minimisation) et dans ce cas nous nous assurons d'un résultat optimal pour les deux critères évalués. Cette approche risque de diminuer le nombre d'arbres d'ordonnancement candidats pour l'extraction finale. C'est une vison pessimiste de la contrainte.
- soit nous prenons systématiquement la valeur moyenne. Cette approche atténue les défauts et qualités des deux approches précédentes.
- soit enfin, nous tenons compte d'annotations données par le concepteur de l'application qui peut indiquer quel est le résultat du test conditionnel qui prédomine sur l'autre pour l'évaluation de ces critères.

Il est toutefois évident que nous risquons ici de perdre le caractère optimal puisqu'il n'est pas obligatoire d'avoir un arbre d'ordonnancement qui optimise un critère simultanément sur tous les comportements d'exécutions. C'est pourquoi la sélection d'une de ces méthodes est laissé au libre arbitre de l'utilisateur de l'outil. Toutefois, l'utilisation de la valeur maximale (dans le cas d'une minimisation) semble être la meilleur approche dans la plupart des cas.

L'application de ces contraintes a posteriori sur le GA permet à l'algorithme d'extraction de sélectionner les marquages optimaux (hormis pour les instructions conditionnelles comme nous venons de voir) au vue de certaines contraintes. Nous pouvons étendre ce principe à d'autres critères plus spécifiques. En effet, nous savons que la prise en compte des tâches conditionnelles engendre une variation du nombre de temps creux pouvant survenir durant l'exécution. Nous pouvons alors utiliser ces temps creux dans l'optique d'un placement judicieux dans les arbres d'ordonnancement.

Parmi les tâches présentes dans les applications temps réel, certaines peuvent survenir à tout instant, ce sont les tâches apériodiques. Or, ces dernières ne peuvent pas être prises en compte directement dans notre outil puisqu'elles n'ont aucun critère temporel quand à leur date de réveil et leur échéance (nous les considérerons comme souples) contrairement aux tâches sporadiques déclenchées. Toutefois, le traitement de ces tâches peut être optimisé par notre méthode en s'inspirant de celles utilisées avec les algorithmes en ligne (serveur à scrutation, serveur sporadique, serveur ajournable, slack stealing *etc...*). La seule possibilité pour ces tâches de s'exécuter est d'utiliser les temps d'inactivité du processeur. Or, grâce à la tâche oisive, nous pouvons tenter de placer au mieux ces temps creux pour permettre d'augmenter le temps de réponse des tâches apériodiques. Pour cela, nous utilisons des heuristiques d'extraction ne portant que sur la tâche oisive. L'idée est de tenter de regrouper tous les temps creux par paquet de façon périodique sur $[0..P]$ de façon à retrouver les méthodes en ligne classiques qui utilisent des serveurs (tâches périodiques critiques rajoutées au système et dédiées au traitement des apériodiques) tout en s'assurant d'un ordonnancement valide des tâches critiques de l'application. Pour exemple, nous pouvons prendre une fonction de pondération χ de type sinusoïdale en fonction du temps dont la période correspond à la période choisie pour simuler la tâche serveur. Puis, nous attribuons à chaque temps creux de hauteur h du graphe d'accessibilité globale, le résultat de $\chi(h)$. Enfin, une exploration ascendante du graphe nous permet d'étiqueter les états du GA qui minimise $\sum_0^P \chi(h)$. Ainsi nous pouvons gérer en traitement de fond

l'exécution de tâches apériodiques (ou de tout autre processus non critique).

De plus, cette méthode d'ordonnancement mixte peut être renforcée par une utilisation adéquate des tâches conditionnelles. En effet, dans des systèmes où la consommation d'énergies est critique, il est souvent nécessaire de proposer des tâches fonctionnant en mode dégradé. Ce mode permet de fournir des résultats moins précis pour le système mais à moindre coût quant à l'activité processeur, en minimisant le code de la tâche et par conséquent sa durée d'exécution. Or, un fonctionnement normal/dégradé peut être modélisé par notre outil grâce aux tâches conditionnelles : un test conditionnel sur une variable globale déterminant le mode en cours permet de choisir l'exécution adéquate. De plus, à l'aide des heuristiques optimisant les traitements en arrière plan, il est devient plus aisé de maîtriser l'exécution des tâches apériodiques pour améliorer leur temps de réponse.

5.3.7 Arbre d'ordonnancement à priorité fixe

Dans la pratique, une grande majorité des OS temps réel utilisent pour ordonnancer les tâches sur le processeur, des priorités fixes qui ont été préalablement attribuées aux tâches. Il semble donc intéressant d'extraire du GA non pas un arbre d'ordonnancement comme pour les techniques précédentes mais une répartition de priorité valide quelques soient les chemins d'exécutions issus des instructions conditionnelles pour chaque tâche.

L'existence d'une telle répartition ne peut être a priori prouvée. Nous avons alors recours à un algorithme de parcours du GA qui ne se termine que lorsque il trouve effectivement une répartition valide ou lorsque toutes les possibilités d'attribution ont été explorées. De ce fait, sachant qu'il y a $n!$ répartitions de priorités pour n tâches, nous pouvons en déduire raisonnablement que cet algorithme ne termine jamais s'il n'existe pas de solution (en considérant une application réaliste). Toutefois, si l'objectif est de vérifier l'existence d'une répartition de priorité parmi un sous ensemble des $n!$, cet algorithme prend tous son sens.

Dans le cas de la vérification de la validité d'une répartition de priorité, il est possible d'apporter des contraintes a priori dès la construction du GA (ces contraintes peuvent toutefois être applicables a posteriori). En effet, deux contraintes définies précédemment permettent de répondre aux caractéristiques des algorithmes à priorités fixes : les arbres d'ordonnancement conservatifs et les contraintes de successeurs. La répartition de priorité peut être mis en évidence dès la construction en modifiant les contraintes de successeurs pour choisir à chaque point d'arrêt (nouvelle activation, prise et libération de ressources, précédence, fin d'exécution) la tâche la plus prioritaire suivant la répartition voulue. Il est également envisageable de permettre aux contraintes de successeur de choisir la prochaine priorité en fonction des précédences et ressources, c'est à dire en prenant en compte des algorithmes de gestion de ressources tel l'héritage de priorité par exemple.

Si le GA a déjà été construit, et que nous souhaitons extraire une répartition valide de priorité pour les tâches, nous adoptons une approche empirique qui comme nous l'avons vu peut ne jamais se terminer. La figure 5.14 illustre le fonctionnement global de cet algorithme.

Nous nous plaçons dans le cas de tâches à départs simultanées périodiques. En partant de l'état initial M_0 du GA, nous choisissons une tâche τ_i à exécuter, celle ci devient la

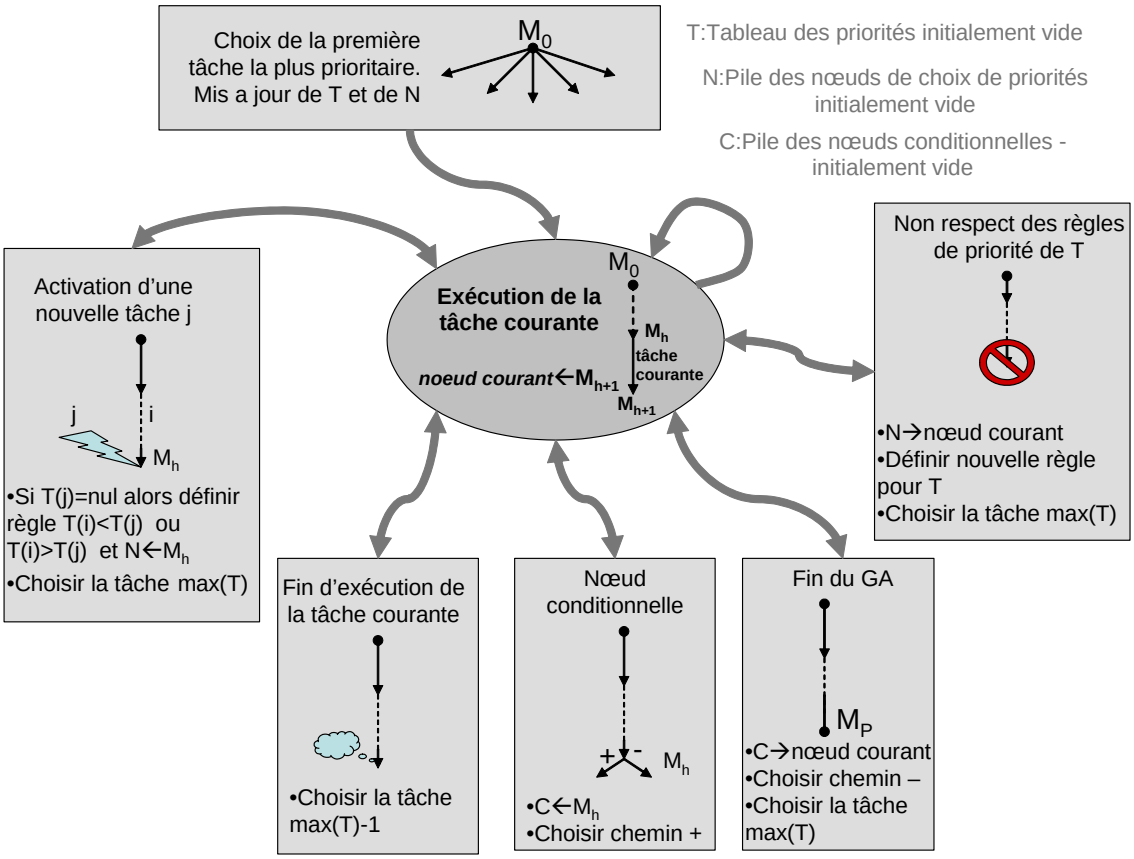

Fig. 5.14: Algorithme schématique d'extraction d'une répartition de priorité valide.

tâche de priorité maximale pour la suite de l'exploration. Ce choix peut être arbitraire ou dicté par une heuristique privilégiant une attribution suivant un algorithme connu et optimal dans son contexte comme RM. Trois cas peuvent alors apparaître :

- la tâche τ_i termine son exécution. Il faut alors définir une priorité pour une autre tâche moins prioritaire que τ_i mais plus prioritaire que toutes les autres. Le noeud courant est alors mémorisé.
- la tâche τ_i exécute une instruction conditionnelle. Il faut mémoriser le noeud du GA pour pouvoir par la suite vérifier que la répartition finale fonctionne également à partir de ce noeud en considérant l'alternative du test conditionnel.
- il n'existe plus de noeud du GA correspondant à une exécution de τ_i alors que celle ci n'a pas fini son exécution. Dans ce cas la tâche τ_i n'est pas la tâche la plus prioritaire et nous devons retourner au noeud précédemment mémorisé pour redéfinir une nouvelle attribution de priorité. τ_i ne peut donc être la tâche la plus prioritaire.

Ce principe de fonctionnement est étendu à l'ensemble du GA et pour toutes les tâches pour pouvoir définir une attribution de priorité complète. Pour cela, nous rajoutons deux étapes :

- une nouvelle activation de tâche survient au noeud du GA en cours. Nous devons

alors consulter notre table de priorité en cours d'élaboration pour choisir la tâche la plus prioritaire à exécuter pour la suite.
- le noeud courant correspond au noeud final du GA. Dans ce cas, une attribution de priorité a été trouvé. S'il ne reste plus de noeuds mémorisés correspondant à un test conditionnel alors cette attribution est valide. Sinon, nous devons vérifier pour tous les noeuds correspondant à une instruction conditionnelle si cette attribution est valide. Si ce n'est pas le cas, l'algorithme retourne au noeud mémorisé correspondant à la dernière évaluation de priorité et modifie cette attribution. L'algorithme redémarre de ce point.

Cet algorithme nécessite stocker toutes les attributions de priorité non valide pour éviter de boucler sur des attributions déjà effectuées ce qui malheureusement, engendre une augmentation de l'espace mémoire nécessaire dans le cas d'application de taille importante.

Nous pouvons comme pour le cas de la recherche par contraintes de successeurs, ajouter un point d'arrêt dans cet algorithme dès qu'il rencontre un noeud correspondant à une prise ou libération de ressources, ou une précédence. Nous rajoutons un processus de traitement des priorités déjà évaluées pour étendre les possibilités aux cas des algorithmes de gestion de ressources et de précédence. Ce processus est exécuté : lorsqu'une anomalie est découverte (l'attribution actuelle de priorité ne fonctionne pas) ou lorsque le noeud courant correspond à une prise/libération de ressources ou émission/réception de messages.

5.4 Exemple de modélisation

Nous considérons une application temps réel constituée de deux tâches. Soit τ_1 une tâche conditionnelle et τ_2 une tâche périodique ne comportant qu'une unique durée d'exécution. Ces deux tâches sont décrites par le pseudo code suivant :

```
TACHE 1:                  TACHE 2:
r=0                       r=0
D=10                      D=15
T=10                      T=15
BEGIN                     BEGIN
 Bloc de durée (1);        Bloc de durée (7);
 IF test THEN             END;
  bloc de durée (3);
 END IF
END;
```

La durée de simulation nécessaire à la validation est de longueur la métapériode P, puisque nous sommes dans le cas de tâches à départs simultanés :

$$P = PPCM(15, 10) = 30.$$

Nous devons ajouter une tâche oisive à cette application pour pouvoir générer des arbres d'ordonnancement non conservatifs. Les paramètres temporels de cette tâche τ_0 sont donnés par $\tau_0 \langle 0, \Delta, 30, 30 \rangle$ avec Δ un intervalle compris entre le nombre de temps creux

cyclique sur la metapériode lorsque la tâche τ_1 s'exécute toujours avec sa plus petite durée (c'est à dire une unité de temps pour le bloc de durée et une pour l'instruction conditionnelle, soit 2 unités de temps) et leur nombre lorsque τ_1 s'exécute toujours avec sa plus longue durée (c'est à dire un bloc de durée de une unité de temps, une instruction conditionnelle de durée 1 et un bloc de durée de 3 unités, soit 5 unités de temps. En calculant la charge processeur dans ces deux cas, nous pouvons obtenir l'intervalle de durée pour la tâche oisive :

$$U_{max} = \sum_{i=1}^{2} \frac{C_{max_i}}{T_i} = \frac{5}{10} + \frac{7}{15} = \frac{29}{30}$$

et

$$U_{min} = \sum_{i=1}^{2} \frac{C_{min_i}}{T_i} = \frac{2}{10} + \frac{7}{15} = \frac{20}{30},$$

Par conséquent, sachant que le nombre de temps creux sur une métapériode se calcule suivant la formule $P(1-U)$, l'intervalle de durée de la tâche oisive est de $\left[30 \cdot \frac{20}{30}, 30 \cdot \frac{29}{30}\right]$, d'où $\Delta = [1, 10]$.

La tâche τ_1 contient une unique instruction conditionnelle, il y a donc 2 comportements possibles d'exécution de la tâche τ_1. Par conséquent d'après la propriété 11, nous en déduisons un nombre de comportements d'exécution sur la métapériode, c'est à dire sur l'intervalle $[0, 30]$de :

$$2^{\frac{30}{10}} = 8.$$

Nous donnons la modélisation par RdP de cette application dans la figure 5.15.

Á partir de cette modélisation nous pouvons appliquer une construction en largeur d'abord pour construire le graphe d'accessibilité. Ce dernier est ensuite réduit par l'algorithme d'épuration pour éliminer les noeuds du GA ne permettant pas de tenir compte de la dualité des instructions conditionnelles. En d'autres termes, nous réduisons le GA à un GA global.

Le graphe d'accessibilité global est par ailleurs contenu d'après la propriété 9 dans un hyperpavé de dimension

$$\sum_{i=1}^{2} \Phi(\zeta_i) + 1 = 2 + 1 + 1 = 4.$$

Le GA global englobe alors l'ensemble des arbres d'ordonnancement valides. Nous pouvons également durant la construction en largeur appliquer une contrainte de successeur pour éliminer les arbres d'ordonnancement dont les exécutions des tâches sont entrelacées inutilement. La figure 5.16 montre le GA Global résultant.

Nous devons extraire un arbre d'ordonnancement du GA global. Pour cela, nous choisissons d'utiliser le critère de minimisation du temps de réponse de la tâche conditionnelle. Nous parcourons alors le GA de M_f vers M_0 et annotons chaque noeud par la valeur du critère conformément à ce qui a été défini dans le chapitre 3. Pour évaluer, les noeuds spécifiques aux instructions conditionnelles, nous devons choisir la valeur à prendre en compte parmi les deux alternatives du test. Dans notre cas, le fait que nous n'ayons pas

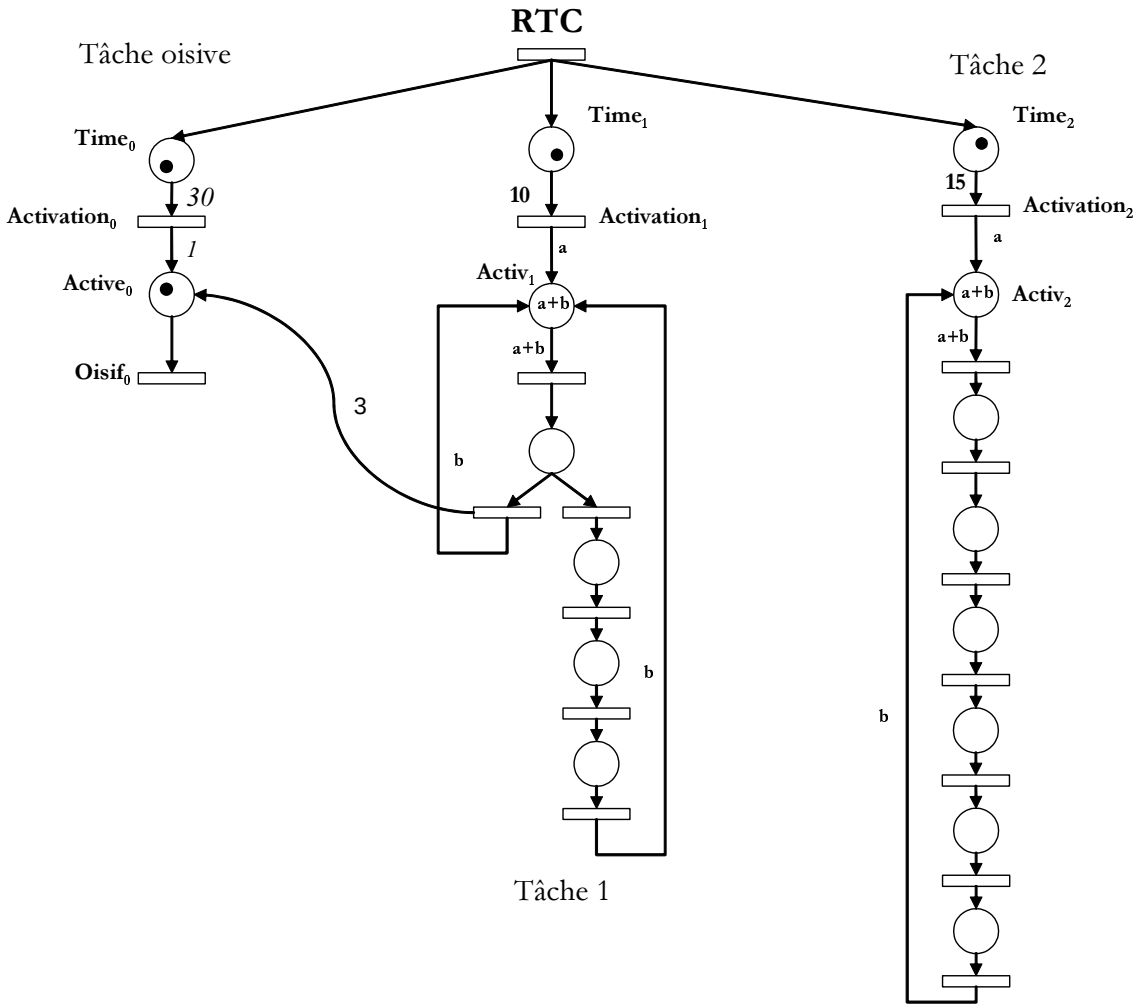

FIG. 5.15: Modélisation par RdP d'un application composée de 2 tâches : $\tau_1 \langle 0, \{2, 5\}, 10, 10\rangle$ et $\tau_2 \langle 0, 7, 15, 15\rangle$. La place processeur n'est pas représentée.

de bloc de durée pour l'alternative ELSE du test, il n'est absolument pas judicieux de prendre systématiquement la plus petite valeur. En effet, celle-ci ne dépend que du placement dans le GA de l'instruction conditionnelle. Ainsi, que ce soit la valeur maximale des alternatives ou la moyenne , le résultat sera ici le même. La figure 5.17 nous montre quel arbre d'ordonnancement sera sélectionné par l'algorithme d'extraction en appliquant un critère de minimisation du temps de réponse pour la tâche conditionnelle τ_1.

5.5 Bibliographie

[CGC96] A. Choquet-Geniet, D. Geniet, and F. Cottet. Exhaustive computation of the scheduled task execution sequences of a real-time application. In *Proc. of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 246–262, Uppsala, Sweden, 1996.

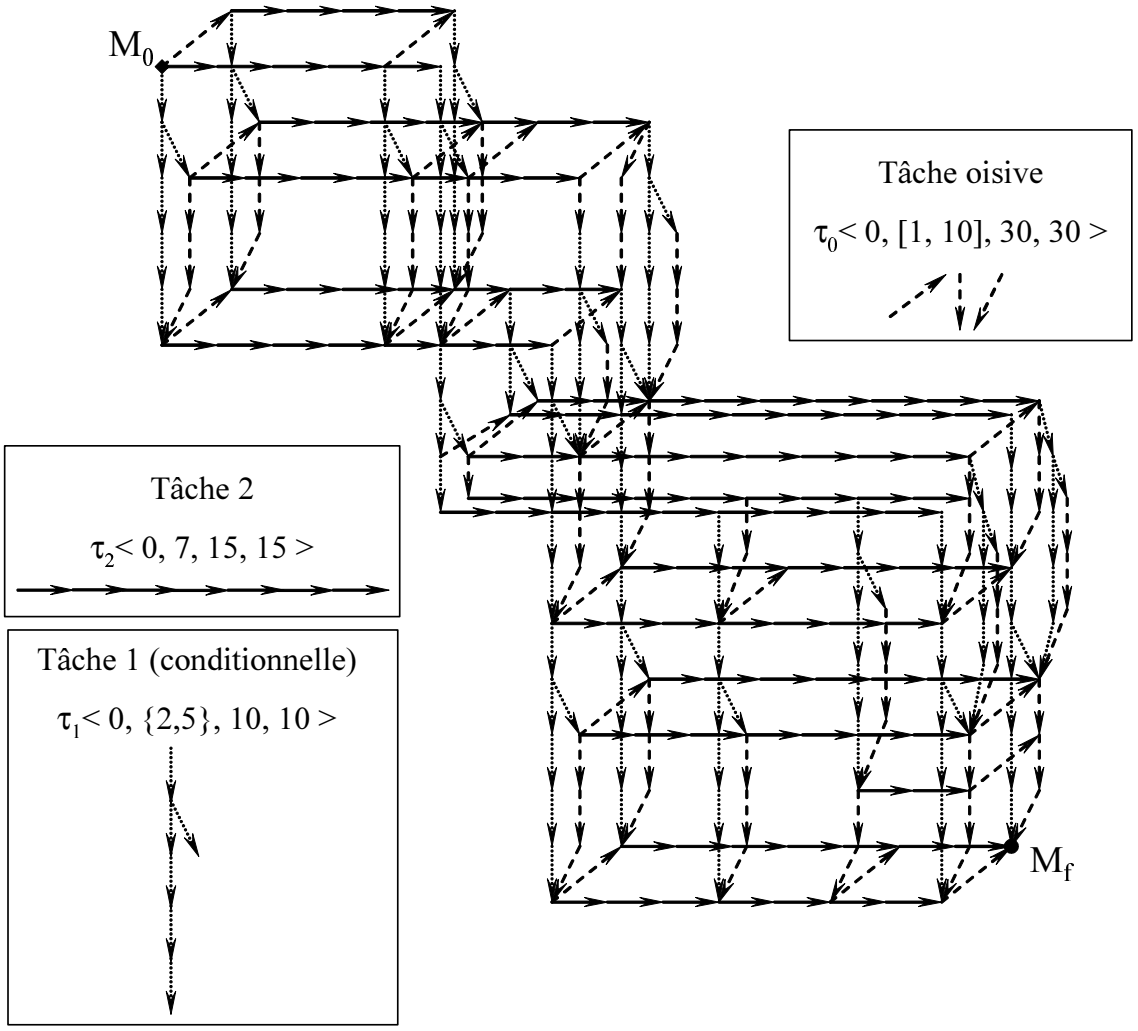

Fig. 5.16: Graphe d'accessibilité global construit par la méthode de construction en largeur d'abord en appliquant une contrainte de successeur. Nous pouvons nous rendre compte de la taille conséquente du GA que peut engendrer un simple exemple d'application composée de 2 taches dont l'une est une tâche conditionnelle.

[GCG02] Emmanuel Grolleau and Annie Choquet-Geniet. Off-line computation of real-time schedules using petri nets. In *Discrete Event Dynamic Systems, DEDS*, volume 12(3), pages 311–333. Kluwer Academic Publishers, Jully 2002.

[GG00] E. Grolleau and A. Choquet Geniet. Off line computation of real-time schedules by means of petri nets. In *Workshop On Discrete Event Systems WODES2000*, Discrete Event Systems : Analysis and Control, pages 309–316, Ghent, Belgium, 2000. Kluwer Academic Publishers.

[Gro99] E. Grolleau. *Ordonnancement temps-réel hors-ligne optimal à l'aide de réseaux de Petri en environnement mono-processeur et multiprocesseur*. PhD thesis, ENSMA - Université de Poitiers, November 1999.

[LG02] Gaëlle Largeteau and Dominique Geniet. Validation temporelle d'applications

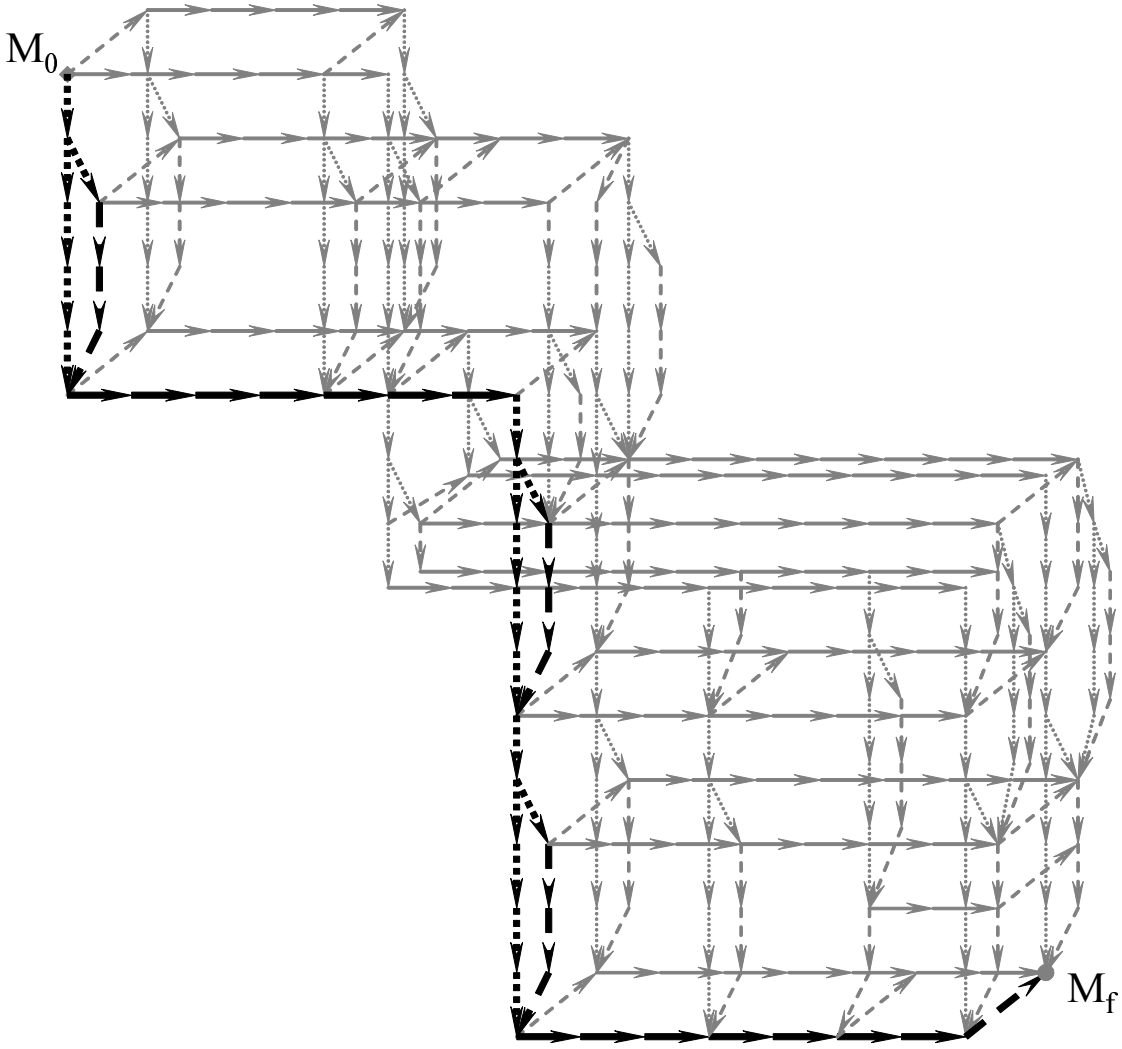

Fig. 5.17: Extraction d'un arbre d'ordonnancement en considérant un critère de sélection par minimisation du temps de réponse.

temps-réel distribuées à contraintes strictes. In *Real-Time Systems*, 2002.

[Ric02] Michaël Richard. *Contribution à la Validation des Systèmes Temps Réel Distribués : Ordonnancement à Priorités Fixes & Placement.* PhD thesis, LISI, ENSMA, Université de Poitiers, Novembre 2002.

[XP90] J. Xu and D. Parnas. Scheduling processes with releases times, deadlines precedence and exclusion relations. In *IEEE transactions on Software Engineering*, volume 16(3), pages 360–369, march 1990.

Conclusion et Perspectives

Nous avons vu en premier lieu la spécificité des systèmes temps réel vis à vis des systèmes informatiques classiques. Nous avons mis en évidence l'importance de la quantification du temps dans les applications temps réel et nous avons alors exhibé les différents modèles temporels de tâches qui les composent ainsi que les différentes mesures pouvant être associées aux tâches. Nous nous sommes ensuite intéressés aux problèmes de l'ordonnancement de tâches et avons énuméré les classes de problèmes d'ordonnancement les plus importants. Nous avons ainsi pu mettre en évidence les difficultés d'ordonnançabilité que peut engendrer un modèle temporel classique (issu de Liu layland) dans les deux courants d'analyse d'ordonnançabilité que sont l'ordonnancement en ligne et hors ligne.

Malgré l'existence d'algorithmes d'ordonnancement en ligne polynomiaux optimaux permettant une validation temporelle analytique des systèmes de tâches, lorsque celles-ci partagent des ressources critiques, le problème de l'ordonnancement devient NP-difficile. 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 ordonnancer les systèmes de tâches très contraints, en effet, les phénomènes de blocage dus à l'attente d'une ressource ne peuvent être évités. Les conditions suffisantes d'ordonnançabilité, se basant sur les temps de blocages maximaux qui peuvent être infligés aux tâches, augmentent de façon artificielle la charge des systèmes de tâches lors de la validation. De plus, même la simulation ne permet pas de valider de tels algorithmes d'ordonnancement sans intégrer toutes les durées de blocage dans les tâches puisque lorsque des ressources critiques sont en jeu, les algorithmes d'ordonnancement ne tiennent compte que des pires durées qui ne correspondent pas obligatoirement au pire cas. L'approche hors ligne s'avère donc la plus indiquée lorsque l'application est fortement couplée (nombreuses contraintes en terme de partage de ressources critiques et communications). Toutefois, en plus de la complexité qu'engendre la recherche de séquence d'ordonnancement, l'ordonnancement hors ligne est également limité par le modèle temporel de tâche. En effet, toutes les tâches peuvent posséder plusieurs comportements d'exécution, certain utilisant des ressources critiques et d'autres possédant une durée d'exécution plus importante. Or, les modèles temporels classiques ne tiennent compte que de la durée d'exécution issue de la pire durée, munie de l'ensemble des ressources utilisées par les différents comportements d'exécutions de la

tâche. Il en ressort un modèle sur-contraint et non représentatif de la réalité, ce qui peut aboutir à tort à la non ordonnançabilité d'un système de tâches.

Notre étude a donc porté sur deux points :

– D'une part, nous avons proposé un nouveau modèle de tâche permettant de ne pas s'appuyer sur la considération du pire cas. À notre connaissance, il s'agit d'un premier travail sur la prise en compte effective des tâches comportant des instructions conditionnelles. Ainsi, si une tâche peut être exécutée avec n comportements d'exécution distincts issus de tests conditionnels, nous choisissons de prendre en compte explicitement ces n comportements dans notre modèle de tâches en intégrant chaque test conditionnel dans la représentation de la durée des tâches. Nous introduisons alors non pas une durée d'exécution C_i déterministe et évaluée en considérant le comportement d'exécution maximale de chaque tâche, mais un multi-ensemble de durées ζ représentant chaque comportement d'exécution d'une tâche. Ce modèle de tâche étendu nous a obligé à revoir la notion d'ordonnancement, en introduisant la notion d'arbre d'ordonnancement. Nous proposons un opérateur de génération des arbres d'ordonnancement de type conservatif. Nous avons introduit des propriétés de vérification du respect des contraintes temporelles de chaque tâche et du partage des ressources pour permettre la construction d'arbres d'ordonnancement valides. Nous avons par la suite proposé deux stratégies d'ordonnancement : l'ordonnançabilité locale vérifie l'ordonnançabilité de chaque configuration d'exécution possible, pour un système de tâches en utilisant les techniques de l'ordonnancement classique. L'ordonnançabilité globale quant à elle repose sur l'existence d'un arbre d'ordonnancement valide. Nous avons montré que, dans le cas de tâches indépendantes, ces deux stratégies sont équivalentes ce qui permet de profiter des résultats déjà établis dans le cas du modèle de tâche classique pour l'ordonnançabilité globale. Enfin, nous avons montré que ces arbres d'ordonnancement pouvaient être étudiés sur une durée de simulation bornée pour leur validation. Nous avons donc mis en évidence la cyclicité des arbres d'ordonnancement en prouvant que la date de rentrée dans le cycle était décroissante au sens large lorsque les durées d'exécution sont globalement décroissantes. Ces résultats permettent d'envisager des études d'ordonnançabilité plus précises en évitant de sur-contraindre la représentation du système de tâches. La prise en compte explicite des instructions conditionnelles permet d'envisager l'ordonnancement de systèmes fonctionnant sous plusieurs régimes (mode normal/mode dégradé) comme pour la gestion de l'économie d'énergie par exemple. Nous avons donc posé les bases formelles du problème de l'ordonnancement dans le contexte des applications à tâches conditionnelles.

– D'autre part, nous avons proposé une méthode d'analyse d'ordonnançabilité et de validation temporelle d'applications temps réel fortement couplées qui utilise une modélisation par réseau de Petri issue des travaux de A. Geniet et E.Grolleau. Après avoir montré comment modéliser ce type d'applications par réseaux de Petri, nous en avons extrait un graphe des marquages. Il a été ensuite nécessaire de filtrer ce graphe pour supprimer les séquences d'ordonnancement non valides de façon à ne garder qu'un graphe d'accessibilité globale formé de l'ensemble des graphes

d'ordonnancement valides. Nous avons montré par la suite comment restreindre la construction de ce graphe en appliquant des contraintes a priori tout en gardant une forte puissance d'ordonnançabilité. L'extraction des graphes d'ordonnancement valides s'effectue également par contraintes portant par exemple sur les temps de réponse. Enfin, nous avons mis en évidence les possibilités qu'offraient cette méthodes en terme de traitement des tâches sporadiques déclenchées.

L'utilisation de tâches conditionnelles ouvre la voie à de nombreuses perspectives. En premier lieu avec notre modélisation par RdP, nous pouvons prendre en compte l'ensemble des comportements d'exécution des tâches d'une application. Or, tous ces comportements engendrent la construction d'un arbre d'ordonnancement pouvant être de taille importante. Il apparaît alors utile de chercher parmi l'ensemble des solutions du graphe d'accessibilité global s'il n'existe pas une distribution de priorité aux tâches (avec ou sans protocoles de gestion de ressources). Ceci revient à trouver un graphe d'ordonnancement dont l'ensemble de ses branches est ordonnancé par une même distribution de priorité. En étendant ce principe, nous pouvons également essayer d'extraire un arbre d'ordonnancement qui peut être transformé en une séquence : chaque branche de l'arbre d'ordonnancement exécute les mêmes tâches (ou la tâche oisive si la tâche en question a terminée son exécution) au même moment. Les tâches conditionnelles permettent également d'étendre les possibilités de modélisation de notre méthode. En effet, pour des systèmes de tâches où le temps alloué au traitement d'une tâche n'a de conséquence que sur la précision du résultat calculé, il convient de maximiser son temps de calcul sans pour autant mettre le système en surcharge. Ces tâches que nous nommons tâches incrémentales, doivent à chaque unité de temps soit retourner un résultat approximatif ou soit continuer leur exécution. Ce choix peut parfaitement être pris en compte par notre modèle de tâches conditionnelles. Toutefois, dans ce cas précis, il faut se focaliser sur l'ordonnançabilité locale et non globale puisque un seul des comportements d'exécution n'est ici utile, celui de durée maximale. Enfin, notre modèle temporel permet d'avoir une description formelle de l'ensemble des comportements du systèmes de tâches. Cette description permet également de prendre en compte l'activation des tâches sporadiques déclenchées lors d'une analyse d'ordonnancement hors ligne. Il pourrait donc être intéressant d'utiliser ce modèle temporel de tâches conditionnelles/sporadiques déclenchées dans une autre approche d'analyse d'ordonnançabilité hors-ligne.

ANNEXES

Table des figures

Index

Bibliographie

[ABD+95] N.C. Audsley, A. Burns, R.I. Davis, K.W. Tindell, and A.J. Wellings. Fixed priority pre-emptive scheduling : an historical perspective. *Real-Time Systems*, 8 :173–198, 1995.

[ABRW91] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard real-time scheduling : the deadline monotonic approach. *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 127–132, 1991.

[ABRW92] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Deadline monotonic scheduling theory. *Real-time Programming*, pages 55–60, 23-26 juin 1992.

[AD92] M. Alabau and T. Dechaize. Ordonnancement temps réel par échéance. In *T.S.I.*, volume 11. n.3, 1992.

[Aud91] N.C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. *Technical Report YCS-164*, 31, nov 1991.

[Bab96] J.P. Babau. *Etude du comportement temporel des applications temps réel à contraintes strictes basées sur une analyse d'ordonnançabilité*. PhD thesis, thèse de doctorat d'informatique, LISI/ENSMA, 1996.

[Bak91] T.P. Baker. Stack-based scheduling of real-time processes. *the Journal of Real-Time Systems*, 3 :67–99, 1991.

[Bar98] S. Baruah. A general model for recurring real-time tasks. In *Proceeding of the Real Time System Symposium, IEEE Computer Society Press*, pages 114–122, Madrid, Spain, 1998.

[Bar03] Sanjoy K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Syst.*, 24(1) :93–128, 2003.

[BCGM99] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized multiframe tasks. *Real-Time Syst.*, 17(1) :5–22, 1999.

[BF86] E. Best and C. Fernandez. Notations and terminology on petri net theory. *Arbeitpapiere der Gesellschaft für Mathematik und Datenverarbeitung MBH 195*, page 29, Jan 1986.

[BG92] Gerard Berry and Georges Gonthier. The esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[BG04] Sanjoy Baruah and Joel Goossens. *Scheduling Real-time Tasks : Algorithms and Complexity. In Handbook of Scheduling : Algorithms, Models, and Performance Analysis, Joseph Y-T Leung.* Chapman Hall/ CRC Press, 2004.

[BHR90] S. Baruah, R. Howel, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic real time tasks on one processor. *Real time systems*, 2 :301–324, 1990.

[Bla76] J. Blazewicz. Scheduling dependant tasks with different arrival times to meet deadlines. *Modeling and performance evaluation of computer systems*, pages 57–65, 1976.

[BM82] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time petri nets. *3rd European Workshop on Applications and Theory of Petri Nets*, Sept 1982.

[BMR90] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. *Proc. 11th IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[BS74] K.R. Baker and Z.S Su. Sequencing with due-dates and early start times to minimize maximum tardiness. In *Naval Research Logistic Quaterly*, volume 21, pages 171–176, 1974.

[But97] G.C. Buttazzo. *Hard real time computing systems : predictable scheduling algorithms and applications.* Kluwer Academic Publisher, 1997.

[BW90] A. Burns and A. Wellings. *Real-Time Systems and their Programming Languages.* Addison-Wesley, 1990.

[CC89] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE TRANS. ON SOFT. ENGENEERING*, 15(10) :1261–1269, 1989.

[CC91] H. Chetto and M. Chetto. An adaptative scheduling algorithm for fault-tolerant real-time systems. In *Software Engineering Journal*, pages 179–186, Mai 1991.

[CD93] H. Chetto and J. Delacroix. Minimisation des temps de réponse des tâches sporadiques en présence des tâches périodiques. *Salon des solutions informatiques temps réel*, pages V39–V51, January 1993.

[CDKM00] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Oronnancement Temps Réel.* Hermès Edition, 2000.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.

[CET01] Samarjit Chakraborty, Thomas Erlebach, and Lothar Thiele. On the complexity of scheduling conditional real-time code. *Proceedings of the Seventh International Workshop on Algorithms and Data Structures (WADS 2001)LNCS 2125*, pages 38–49, 2001.

[CG06] Annie Choquet-Geniet. *Les réseaux de Petri, un outil de modélisation*, volume ISBN=2100491474. Dunod, 2006.

[CGC96] A. Choquet-Geniet, D. Geniet, and F. Cottet. Exhaustive computation of the scheduled task execution sequences of a real-time application. In *Proc. of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 246–262, Uppsala, Sweden, 1996.

[CGG04] A. Choquet-Geniet and E. Grolleau. Minimal schedulability interval for real time systems of periodic tasks with offsets. In *Theoretical of Computer Sciences*, volume 310, pages 117–134, March 2004.

[CGVN93] A. Choquet-Geniet and G. Vidal-Naquet. *Réseaux de Petri et systèmes parallèles*. Ed. Armand Colin, 1993.

[CHB79] R.H. Campbell, K.H. Hurton, and G.G. Belford. Simulations of fault-tolerant real-time deadlin mechanisms. In *Proceedings of FCTS'9*, pages 95–101, Janvier 1979.

[Che02] Albert M. K. Cheng. *Real Time Systems, Scheduling Analysis and Verification*. John Wiley & Sons, 2002.

[Chr83] P. Chrétienne. *Les réseaux de Petri temporisés*. PhD thesis, Thèse dŠEtat, Université Paris VI, Juin 1983.

[CL88] J.Y. Chung and J.W.S Liu. Algorithms for scheduling periodics jobs to minimize average error. In *9th IEEE RTSS*, pages 142–152, Décembre 1988.

[CL90] M. Chen and K. Lin. Dynamic priority ceilings : a concurrency protocol for real time systems. *Real time systems*, no. 2 :235–346, 1990.

[CLB99] M. Caccamo, G. Lipari, and G. Buttazzo. Sharing ressources among periodic and aperiodic taskc with dynamic deadlines. *In proceeding of th* 20th *IEEE Real Time System Symposium*, 1999.

[CNR88] G.D.R.T.R CNRS. Le temps réel. *Technique et Science Informatiques*, 1988.

[Col01] A. Colin. *Estimation de temps d'exécution au pire cas par analyse statique et application aux systèmes d'exploitation temps-réel*. PhD thesis, niversité de Rennes I, October 2001.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. *In proceeding of the third annual ACM symposium on Theory of Computing*, 1971.

[Coo83] Stephen A. Cook. An overview of computational complexity. *Communication of ACM*, 26 :400–408, 1983.

[Cot99] F. Cottet. Etude de l'application temps réel embarquée : mission pathfinder. *actes de l'école d'été temps réel, ETR99*, pages 109–117, 13-16 sept 1999.

[CP00] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2) :249–274, May 2000.

[Der74] M. Dertouzos. Control robotics : the procedural control of physical processors. *Proc. IFIP Congress*, pages 807–813, 1974.

[DM89] M.L. Dertouzos and A.K. Mok. Multiprocessor on-line scheduling of hard real-time tasks. *IEEE Transactions on Software Engineering*, 15(12) :1497–1506, Dec. 1989.

[DTB93] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. *In proceeding of IEEE Real Time System Symposium*, pages 22–231, 1993.

[Ell97] Jean Pierre Elloy. Systèmes réactifs synchrones et asynchrones. In *Applications, Réseaux et Systèmes – École d'été temps réel'99*, pages 43–51, Futuroscope, Septembre 1997.

[Foh94] G. Fohler. *Flexibility in statically scheduled hard real-time sys-ems.* PhD thesis, University of Wien, April 1994.

[Gal97] L. Gallon. *Le modèle réseaux de Petri temporisés stochastiques : extensions et applications.* PhD thesis, Université Paul Sabatier, Dec 1997.

[GB95] T.M. Ghazalie and Theodore P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9(1) :31–67, 1995.

[GCG02] Emmanuel Grolleau and Annie Choquet-Geniet. Off-line computation of real-time schedules using petri nets. In *Discrete Event Dynamic Systems, DEDS*, volume 12(3), pages 311–333. Kluwer Academic Publishers, Jully 2002.

[Gen00] Annie Geniet. *Systèmes paralléles et temps réel : analyse à l'aide de modèles formels.* PhD thesis, Habilitation a diriger des recherches, LISI/ENSMA, Decembre 2000.

[GG00] E. Grolleau and A. Choquet Geniet. Off line computation of real-time schedules by means of petri nets. In *Workshop On Discrete Event Systems WODES2000*, Discrete Event Systems : Analysis and Control, pages 309–316, Ghent, Belgium, 2000. Kluwer Academic Publishers.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractibility : a guide to the theory of NP-completness.* W H Freeman & Co, 1979.

[GLB+02] J. Gustafsson, B. Lisper, N. Bermudo, C. Sandberg, and L. Sjöberg. A prototype tool for flow analysis of c program. In *2nd Intl. Workshop on WCET Analysis 14th Euromicro Conference on Real- Time Systems*, June 2002.

[Gra69] R. Graham. Bounds on multiprocessing timing anomalies. In *SIAM Journal of Applied Mathematics 17, no. 2*, pages 416–429, 1969.

[Gro99] E. Grolleau. *Ordonnancement temps-réel hors-ligne optimal à l'aide de réseaux de Petri en environnement mono-processeur et multiprocesseur.* PhD thesis, ENSMA - Université de Poitiers, November 1999.

[Har87] P.K. Harter. Response time in level structured systems. *ACM Transactions on Computer Systems*, 5(3) :232–248, 1987.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[Hen75] R Henn. *Deterministic modelle fur die prozessorzuteilung in einer harten realzeit-umgebung.* PhD thesis, Technical university, Munich, 1975.

[HM95] C. Hanen and A. Munier. *Cyclic scheduling on parallel processors : An Overview*, volume Scheduling theory and its applications, P. Chretienne et al., Chap 9. John Wiley & Sons, 1995.

[HR95] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with, 1995.

[HSRW98] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, Discrete Event Systems : Analysis and Control, pages 12–21. IEEE, 1998.

[HSSM68] A. Holt, H. Saint, R. Shapiro, and S. Marshall. Final report of the information system theory project. *Tech. Rep. RADC-TR-68-305, Rome Air Development Center, Griffis Air Force Base*, page 352, Sept 1968.

[HST97] H. Hansson, M. Sjodin, and K. Tindell. Guaranteeing real-time traffic through an atm network. *In Proc. of the 30'th Hawaii International Conference on System Sciences, IEEE Computer Society Press*, 5 :44–53., January 1997.

[IFS03] D. Isovic, G. Fohler, and L. Steffens. Timing constraints of mpeg-2 decoding for high quality video : misconceptions and realistic assumptions. *In Proceedings of the 15th Euromicro Conference on Real-Time Systems*, June 2003.

[IFS04] Damir Isovic, Gerhard Fohler, and Liesbeth Steffens. Real-time issues of mpeg-2 playout in resource constrained systems. *International Journal on Embedded Systems*, 1, issue 2(ISSN 1740-4460), 6 2004.

[Jac55] J.R. Jackson. Scheduling a production line to minimize maximum tardiness. *Management Sciences Research Project*, 1955.

[Jen81] K. Jensen. Coloured petri nets and the invariant-method. *Theoretical Computer Science*, 14 :317–336, 1981.

[Jen94] K. Jensen. An introduction to the theoretical aspects of coloured petri nets. *A Decade of Concurrency, in Lecture Notes in Computer Science*, 803 :230–272, 1994.

[Jen96] K. Jensen. Coulored petri nets, basic concepts, analysis methos and practical use. *2nd edition, Monographs in theoretical Computer Science*, 1 :234, 1996.

[Jen98] K. Jensen. An introduction to the practical use of coloured petri nets. *Lectures on Petri Nets : Applications, in Lecture Notes in Computer Science*, 1492 :237–292, 1998.

[JLKD86] R. Janicki, P.E. Lauer, M. Koutny, and R. Devillers. Concurrent and maximally concurrent evolution of nonsequential systems. *Theoretical Computer Science*, 43(2-3) :213–238, 1986.

[Jos85] M. Joseph. On a problem in real-time computing. *Information Processing Letters*, 20(4) :173–177, 1985.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *Computer Journal*, 29(5) :390–395, 1986.

[JP96] Beauvais. J.-P. *Etude d'algorithmes de placement de tâches temps réel périodiques complexes dans un système réparti*. PhD thesis, Thèse de doctorat de l'Ecole Centrale de Nantes et de l'Université de Nantes., 1996.

[JSM91] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *In Proceedings of the 12 th IEEE Symposium on Real-Time Systems*, pages 129–139, Decembre 1991.

[Jua99] G. Juanole. Modélisation et évaluation du protocole mac du réseau can. *actes de ETR'99*, pages 187–200, 13-16 sept 1999.

[Kai01] C. Kaiser. *Description et critique d un système temps réel pour le suivi d un laminoir : Robustesse et potentiel d'évolutivité*, volume 20(1). Hermes Science, 1901.

[Kai82] C. Kaiser. Exclusion mutuelle et ordonnancement par priorité. *Technique et Science Informatiques*, 1982.

[Kop92] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *12th Int. Conf. On Distributed Computing Systems*, pages 460–467, Japon, June 1992.

[Kop97] Hermann Kopetz. *Real-Time Systems : Design Principles for Distributed Embedded Applications. Series : The International Series in Engineering and Computer Science, Vol. 395*, volume ISBN : 0 7923 9894 7. Kluwer Academic Publishers, 1997.

[Kra85] S. Krakowiak. *Principes des systèmes d'exploitation.* Dunod, 1985.

[KRP+93] M. Klein, T. Rayla, B. Pollak, R. Obenza, and M.G. Harbour. *A practitionner's handbook for real time systems analysis.* Kluwer Academic Publisher, 1993.

[KS86] E. Klingerman and A.D. Stoyenko. Real time euclid : a language for reliable real time systems. *IEEE Trans. softw. Eng.*, 12(9) :941–949, 1986.

[KS99] C. Kaiser and G. Stoffel. Systeme d acquisition et d analyse en temps reel des signaux d un laminoir. *Rapport scientifique CEDRIC*, 1999.

[KSSK96] H. Kaneko, J.A Stankovic, S. Sen, and K.Ramamritham. Integrated scheduling of multimedia and hard real time tasks. *Technical report UMCS 1996-045*, Computer Science, 1996.

[Lab74] J. Labetoulle. Un algorithme optimal pour la gestion des processus en temps réel. *Revue Française d'Automatique, Informatique et Recherche Opérationnelle*, pages 11–17, 1974.

[LCG04] Gaëlle Largeteau, Bernard Chauvière, and Dominique Geniet. Une approche géométrique de la validation d'applications temps réel. In *Real-Time Systems*, 2004.

[Leh90] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. *Proc. IEEE Real-Time Systems Symposium*, pages 201–209, 1990.

[LG02] Gaëlle Largeteau and Dominique Geniet. Validation temporelle d'applications temps-réel distribuées à contraintes strictes. In *Real-Time Systems*, 2002.

[LG05] Gaëlle Largeteau and Dominique Geniet. Discrete geometry applied in hard real-time systems validation. *Proc. of 12th Discrete Geometry for Computer Imagery, Lecture Notes in Computer Science*, 3429 :23–33, Springer-Verlag 2005.

[Liu00] J. Liu. *Real-time systems*. Prentice Hall, 2000.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for mutltiprogramming in real-time environnement. *Journal of the ACM*, 20(1) :46–61, 1973.

[LM80] J.Y.T. Leung and M.L. Merill. A note on preemptive scheduling of periodic real-time tasks. In *Information Processing Letters 11(3)*, pages 115–118, November 1980.

[LRT92] J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft aperiodic tasks in fixed priority preemptive system. *Proceeding of the Real Time System Symposium, IEEE*, pages 110–123, december 1992.

[LSD89] J.P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm : exact characterization and average case behavior. *Proc. 10th Real-Time Systems Symposium*, pages 166–171, december 1989.

[LSS87] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced periodic responsiveness in hard real time environments. In *Proceeding of the RTSS, IEEE*, pages 261–270, San Jose, December 1987.

[LSST91] J.P. Lehoczky, L. Sha, J.K. Strosnider, and H. Tokuda. Fixed priority scheduling theory for hard real-time systems. *in Foundations of Real-Time Computing : Scheduling and resource management*, pages 1–30, 1991.

[LW82] J. Leung and J. Whitehead. On the complexity of fixed priority scheduling of périodic real-time tasks. *Performance Evaluation*, pages 237–250, 1982.

[MA98] Y. Manabe and S. Aoyagi. A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling. *Real-Time Systems*, 14(2) :171–181, Mars 1998.

[MC96] A. K. Mok and D. Chen. A multiframe model for real-time tasks. In *RTSS '96 : Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 22, Washington, DC, USA, 1996. IEEE Computer Society.

[MF76] P.M. Merlin and D.J. Farber. Recoverability of communication protocols - implications of a theoretical study. *IEEE Transactions on Communications*, pages 1036–1043, Sept 1976.

[Mok83] A.K. Mok. *Fundamental design problems for the hard real time environments*. PhD thesis, MIT, 1983.

[MPS78] M. Moalla, J. Pulou, and J. Sifakis. Réseaux de petri synchronisés. *Journal RAIRO, Automatic Control Series*, vol. 12, n2 :103 :130, 1978.

[MR02] T. Mitra and A. Roychoudhury. A framework to model branch prediction for wcet analysis. In *2nd Intl. Workshop on WCET Analysis 14th Euromicro Conference on Real- Time Systems*, June 2002.

[Mur89] T. Murata. Petri nets : properties, analysis and applications. *Invited Paper, Proc. Of the IEEE*, 17(4) :541–580, 1989.

[PB00] P. Puschner and A. Burns. Guest editorial : A review of worst-case execution-time analysis. *Real-Time Systems*, 18 :115–128, 2000.

[PC02] S. Pailler and A. Choquet-Geniet. Ordonnancement temps réel comportant des tâches à durées variables. In Tecknea, editor, *RTS Embedded System*, pages 151–172, Paris, France, 2002.

[PC04] S. Pailler and A. Choquet-Geniet. Off-line scheduling of real time aplications with variable duration tasks. In Kluwer Academic Publisher, editor, *Workshop on Discrete Event Systems, WODES2004, Discrete Event Systems Analysis and Control*, Reims, France, September 2004.

[Pet62] C.A. Petri. Kommunikation mit automaten. *Bonn Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, English translation, 1966*, pages Vol.1, Suppl. 1, 1962.

[Pet80] 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 n4*, pages 21–27, 1980.

[Pet81] J.L. Peterson. Petri nets theory and the modeling of systems. *Prentice-Hall, Englewood Cliffs*, page 290, 1981.

[PK89] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2) :159–176, 1989.

[PN98] P. Puschner and R. Nossal. Testing the results of static worst case execution time analysis. In *IEEE Real Time Systems Symposium*, dec 1998.

[Pua05] I. Puaut. Méthodes de calcul de wcet (worst case execution time) etat de l'art. *Ecole d'été Temps Réel*, 4 :165–175, Septembre 2005.

[Ram74] C. Ramchandani. *Analysis of asynchronous concurrent systems by timed Petri nets*. PhD thesis, MIT, Cambridge, Feb 1974.

[RC99] P. Richard and F. Cottet. Ordonnancement temps réel monoprocesseur avec contraintes de précédence simples et généralisées. *Tech. Report 99-002*, 1999.

[RCK01] Pascal Richard, Francis Cottet, and Claude Kaiser. Précédences généralisées et ordonnançabilité des tâches de suivi temps réel d'un laminoir. *Journal Européen des Systèmes Automatisés*, 35 (9) :1055–1071, 2001.

[RCM96] I. Ripoll, A. Crespo, and A.K. Mok. Improvement in feasibility testing for real-time tasks. *Journal of Real-Time Systems*, pages 19–39, 1996.

[RCR01] P. Richard, F. Cottet, and M. Richard. Online scheduling of real time distributed computers with complex communication constraints. 7^{th} *International Conference on Engineering of Complex Computer Systems*, pages 23–24, 2001.

[Ric02] Michaël Richard. *Contribution à la Validation des Systèmes Temps Réel Distribués : Ordonnancement à Priorités Fixes & Placement*. PhD thesis, LISI, ENSMA, Université de Poitiers, Novembre 2002.

[Ric03] Pascal Richard. Analyse du temps de réponse des systèmes temps réel. *Actes de l'École d'Été Temps Réel*, pages 241–262, 2003.

[Rou99] O. Roux. Les langages reactifs synchrones et asynchrones. *Chapitre des Techniques de l'Ingenieur, Traite Controle et Mesure*, 11,(4) :448–471, december 1999.

[RS02] C. Rochange and P. Sainrat. Difficulties in computing the wcet for processors with speculative execution. In *2nd Intl. Workshop on WCET Analysis 14th Euromicro Conference on Real- Time Systems*, June 2002.

[SB94] M. Spuri and G Buttazzo. Efficient aperiodic service under earliest deadline scheduling. *In proceeding of th* 15^{th} *IEEE Real Time System Symposium*, pages 2–21, 1994.

[SB96a] M. Spuri and G Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *the Journal of Real-Time Systems*, 10 :179–210, 1996.

[SB96b] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *The Journal of real time systems*, 10 :179–210, 1996.

[SBS95] M. Spuri, G. Buttazzo, and F. Sensini. Robust aperiodic sceduling under dynamic priority systems. *In proceeding of IEEE Real Time System Symposium*, december 1995.

[SCE90] M. Silly, H. Chetto, and N. Elyounsi. An optimal algorithm for guaranteeing sporadic tasks in hard real time systems. *IEEE Symposium on Parallel and Distribued systems*, pages 578–585, Dec 1990.

[Ser72] O. Serlin. Scheduling of time critical processes. *proc. Spring Joint Computers Conference*, pages 925–932, 1972.

[Sil94] M. Silly. Un algorithme d'ordonnancement des tâches sporadiques pour les systemes temps réel. *APII,*, 28, nř2 :179–205, 1994.

[SRL87] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inherence protocols. *Tech. report CMU-CS-87-181*, December 1987.

[SRL90] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inherence protocols : An approach to real time synchronization. *IEEE Transactions on Computers*, 39 :175–185, December 1990.

[SS93] L. Sha and S.S. Sathaye. A systematic approach to designing distributed real-time systems. *IEEE Computer*, 26 :68–78, sept 1993.

[SS94] M. Spuri and J. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. *IEEE Transactions on computer*, 12 :1407–1412, 1994.

[SSDB94] J.A. Stankovic, M. Spuri, M. DiNatale, and G. Buttazzo. Implications of classical scheduling results for real time systems. *IEEE computer*, vol 28-N6, 1994.

[SSL89] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic task scheduling for hard real time systems. *The journal of Real Time Systems*, 1 :27–60, 1989.

[Sta88] J.A. Stankovic. Misconception about real -time computing. In *IEEE Computer Magazine*, volume 10, pages 0–19. 21, 1988.

[Sta90] 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 :418–432, 1990.

[Tin76] K.W. Tindell. *Fixed priority scheduling of hard real time systems*. PhD thesis, University of York, 1976.

[TLS96] T.S. Tia, J.W.S Liu, and M. Shankar. Algorithms and optimality of scheduling soft aperiodique requests in fixed priority preemptive systems. *The journal of Real Time Systems*, 10, nř1 :23–43, 1996.

[XP90] J. Xu and D. Parnas. Scheduling processes with releases times, deadlines precedence and exclusion relations. In *IEEE transactions on Software Engineering*, volume 16(3), pages 360–369, march 1990.

[XP92] J. Xu and D. Parnas. Pre-runtime scheduling of processes with exclusion relations on nested or overlapping critical sections. In *Phoenix Conference on Computers and Communications*, pages 6.4.7.1–6.4.7.9, April 1992.

[ZS94] Q. Zheng and K.G. Shin. On the ability of establishing real-time channels in point-to-point packet-switched networks. *IEEE Transactions on Communications*, pages 42(2,3,4), 1994.

RÉSUMÉ :

Nous considérons le problème de l'ordonnancement hors ligne d'applications Temps Réel multitâches dans le contexte où les tâches peuvent comporter des instructions conditionnelles. Nous redéfinissons le modèle temporel de tâche pour prendre en compte explicitement les instructions conditionnelles, en étendant le modèle initial de Liu-Layland. Nous reformulons le problème de l'ordonnançabilité pour des tâches indépendantes et mettons en évidence l'ordonnançabilité globale et locale. Puis, nous étudions l'impact de la présence d'instructions conditionnelles sur les durées nécessaires de simulation. Enfin, nous proposons une méthode d'analyse d'ordonnançabilité fondée sur une modélisation par réseaux de Petri qui étend la méthode proposée par E. Grolleau dans sa thèse. L'ajout de tâches conditionnelles dans cette modélisation permet d'intégrer explicitement les différents comportements d'exécution des tâches et de prendre en charge l'activation des tâches sporadiques.

ABSTRACT :

We consider the off line scheduling problem of Real Time applications in the context where the tasks of the applications can have conditional instructions. We redefine the temporal model of task to take the conditional instructions explicitly into account, by extending the initial model of Liu-Layland to that of conditional task. Then, we transform the concept of scheduling into scheduling tree. We reformulate the problem of schedulability of independent tasks and we show the global and local schedulability. Then, we study how conditional task can change the laps of time necessary for the simulation of the application. Finally, we propose a method of schedulability analysis which uses a Petri nets modeling which extends the method suggested by Emmanuel Grolleau in his thesis. We can explicitly take every different behaviors of task into account with the conditional task approch and also deal with sporadic tasks released by particular behaviors of conditional task.

SECTEUR DE RECHERCHE : Informatique

MOTS-CLÉS : Systèmes temps réel, Modélisation par Réseaux de Petri, tâches conditionnelles, tâches sporadiques, ordonnancement global, ordonnancement local

LISI-ENSMA
Téléport 2
1, avenue Clément Ader
BP40109
86 961 Futuroscope Chasseneuil Cedex