Ordonnancement Temps Réel d'applications comportant des tâches à durées variables

Stéphane PAILLER - Annie CHOQUET-GENIET
Laboratoire d'Informatique Scientifique et Industrielle
École Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2 - 1 Avenue Clément Ader
B.P. 40109 - F 86961 Futuroscope Cedex
{stephane.pailler, ageniet}@ensma.fr

Résumé : Nous proposons une méthodologie d'ordonnancement hors ligne d'applications temps réel comportant des tâches à durées variables, les variations provenant de la présence d'instructions conditionnelles dans le corps des tâches. Après avoir adapté le modèle temporel de tâches à ce contexte, nous modélisons ces applications à l'aide de réseaux de Petri autonomes fonctionnant sous la règle de tir maximal et munis d'ensemble terminaux. Nous définissons deux concepts d'ordonnançabilité : l'ordonnançabilité locale et l'ordonnançabilité globale et nous définissons la notion d'arbre d'ordonnançabilité. Puis, nous montrons comment obtenir les arbres d'ordonnancement à partir des graphes d'ordonnançabilité globale.

Mots clés : Durée variable, instruction conditionnelle, ordonnançabilité locale, ordonnançabilité globale, arbre d'ordonnancement, réseau de Petri, ordonnancement hors ligne.

Correspondance : S. Pailler

1. Introduction

Nous nous intéressons au problème de l'ordonnancement d'applications temps réel comportant des tâches à durées variables, ces durées pouvant être tributaires de l'état du procédé contrôlé par l'application, de la position d'un paramètre par rapport à un seuil, induisant des traitements différents selon que l'on se situe au dessus ou au dessous du seuil…… Des tâches à durées imprécises de nature différente ont déjà été étudiées, citons - le modèle à multi représentations [CHB79, Che91], 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 et - 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. L'approche que nous adoptons est totalement différente, puisque nous considérons des fluctuations de durées liées à la présence d'embranchements dans le code de la tâche. Il s'agit donc de variations structurelles, par opposition aux variations conceptuelles évoquées ci-dessus.

Nous considérons des applications constituées de tâches périodiques à contraintes strictes. Chacune est caractérisée par quatre paramètres temporels [LL73, SSR98] : - sa date de premier réveil, - sa période, - son délai critique (qui est la taille de la fenêtre temporelle à l'intérieur de laquelle chaque instance de la tâche doit s'exécuter) et - sa durée d'exécution. Cette durée est généralement supposée connue et déterministe, ce déterminisme s'expliquant par une vue du “pire cas” : pour chaque tâche, on considère la durée maximale d'occupation du processeur nécessaire à son exécution. Cependant, au cours de la vie de l'application, certaines instances de la tâche peuvent avoir des durées plus courtes. Un premier problème posé par cette vue de la pire durée est celui de la non stabilité des algorithmes d'ordonnancement en ligne classiques (ED [LL73], RM [LL73, LSD89], LL [MD78]) : ceci traduit le fait que si on utilise des ressources critiques, le pire cas en terme d'ordonnancement ne correspond pas nécessairement au cas où les durées effectives sont maximales. La diminution de la durée d'une tâche peut induire des fautes temporelles. Ceci provient de ce que ces algorithmes sont conservatifs (le processeur ne reste pas inactif s'il y a des tâches prêtes), et non clairvoyants (ils n'ont connaissance que de l'état instantané de l'application, mais ne connaissent pas l'application dans son ensemble, en particulier ils ne savent pas quand auront lieu les prochains réveils de tâches). Par ailleurs, cette approche peut correspondre à une vision non réaliste de l'application. En effet, une cause de fluctuation des durées d'exécution est la présence au sein du corps de la tâche, d'instructions conditionnelles dont les branches ont des durées distinctes. L'optique du pire cas transforme, du point de vue temporel, les blocs conditionnels en blocs de durée fixe, égale à la durée de la plus longue des branches [Bab96], qui de plus incluent l'intégralité des primitives temps réel présentes dans l'une ou l'autre des branches. La spécificité des instructions conditionnelles se trouve ainsi gommée, et la gestion conditionnelle des ressources et des synchronisations est oblitérée. De ce fait, on travaille avec un modèle de l'application beaucoup plus contraint que ne l'est l'application elle-même. Il s'ensuit que la validation de l'application en vue de la mise en œuvre d'un algorithme d'ordonnancement en ligne couplé avec un protocole de gestion de ressources (PHP [Kai82, SRL90], PCP [SRL90, CL90], SRP [Bak91]) devient très difficile, les conditions suffisantes vérifiées étant très largement surévaluées par rapport aux conditions réellement nécessaires (car on prend en compte des blocages qui en fait ne se produiront jamais simultanément).

Même si l'on s'oriente vers des techniques d'ordonnancement hors ligne, qui peuvent être préférables lorsque l'on considère des applications fortement couplées, car elles sont plus puissantes, la vue du pire cas conduit à des résultats non réalistes : on ne produit pas une

séquence correspondant à une exécution réelle, et là encore, on prend en compte des contraintes beaucoup plus fortes que les contraintes effectives.

Notre objectif est de proposer une vision plus réaliste de l'application, en faisant apparaître de manière explicite les blocs conditionnels. Nous nous appuierons sur la méthodologie basée sur une modélisation par réseaux de Petri proposée dans [CGC96, GCG00] pour les applications composées de tâches périodiques à durées fixes. Cette méthodologie permet d'étudier des applications comportant des tâches à départs simultanés ou différés, utilisant des ressources critiques mono ou multi instances, éventuellement en mode lecteur/écrivain, et communiquant. Nous proposons d'étendre cette méthodologie pour prendre en compte les tâches comportant des instructions conditionnelles.

Dans un premier temps, nous présentons le modèle de tâches étendu, et nous redéfinissons le problème de l'ordonnancement dans ce contexte. Nous étendons la notion de séquence d'ordonnancement, en introduisant la notion d'arborescence d'ordonnancement : il s'agit d'un arbre dont chaque branche est une séquence valide de l'application où les blocs conditionnels sont remplacés par une de leurs branches. Nous définissons ensuite les notions d'ordonnançabilité locale et globale. L'ordonnançablité locale consiste à étudier indépendamment l'ordonnançabilité des applications obtenues en ne conservant pour chaque bloc conditionnel qu'un unique chemin. L'ordonnançabilité globale consiste à déterminer l'existence d'au moins une arborescence respectant toutes les contraintes temporelles. Nous montrons que les deux problèmes ne sont pas équivalents. Dans une deuxième partie, nous présentons succintement le modèle utilisé pour les applications à durée fixe : il s'agit d'une modélisation par un réseau de Petri avec marquages colorés, dont la puissance d'expression a été augmentée d'une part par un fonctionnement sous la règle de tir maximal [Sta90] et d'autre part par l'adjonction d'un ensemble terminal [VVN81]. Nous introduisons également la tâche oisive, qui est une tâche modélisant l'inactivité du processeur, et qui est nécessaire pour la production de séquences non conservatives. Nous présentons ensuite l'extension du modèle. La prise en compte des instructions conditionnelles dans le corps des tâche se fait de manière classique. Nous revenons par contre sur la tâche oisive, dont la durée devient elle aussi variable, bien qu'elle ne contienne pas d'instructions conditionnelles. Nous montrons comment nous réglons le problème de sa durée.

Enfin, dans une dernière partie, nous montrons comment réaliser l'analyse du réseau de Petri. Dans le modèle à durée fixe, nous avions établi que les chemins du graphe des marquages correspondaient aux séquences valides. Nous montrons ici que nous pouvons extraire les arborescences valides du graphe des marquages, et nous présentons l'algorithme d'extraction.

2. Le modèle de tâche étendu

2.1. Le modèle temporel

Nous considérons des applications temps réel composées de tâches périodiques. Elles peuvent utiliser des ressources non préemptives et communiquer entre elles. Ces tâches sont constituées de blocs fonctionnels écrits dans un langage de haut niveau et de primitives temps réel (prise et libération de ressources, émission et réception de messages).

Lorsque les durées des tâches sont connues et fixes (ou bien si l'on a adopté l'optique de la pire durée), la modélisation temporelle adoptée est celle issue de Liu et Layland [LL73] : chaque tâche est décrite par quatre paramètres temporels dont la signification est illustré par la figure 1 :

- r_i, date de première activation,

- C_i, pire durée d'exécution,
- R_i, délai critique,
- T_i, période d' activation.

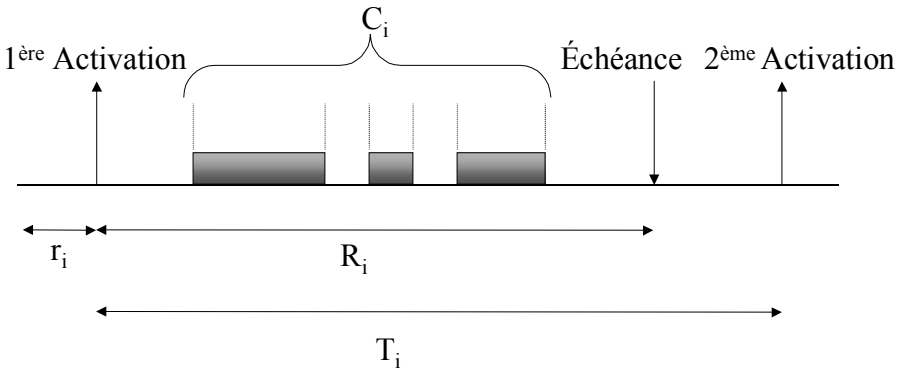


Figure 1. - Paramètres temporels d'une tâche $\tau_i = (r_i, C_i, R_i, T_i)$.

Notre objectif est d'affiner la description des blocs fonctionnels, que nous supposons constitués également d'instructions conditionnelles, ce qui nécessite de reprendre la modélisation temporelle des tâches.

Nous avons étendu le modèle de Liu-Layland de façon à faire apparaître toutes les durées des branches conditionnelles : une tâche est représentée par trois paramètres déterministes (la date de première activation, le délai critique et la période) et par un **multi ensemble D** de durées, chacune correspondant à un comportement possible de la tâche. Notons qu'en l'absence d'instructions conditionnelles, le multi ensemble des durées est de cardinal 1, on retrouve le modèle usuel. La figure 2 illustre la différence entre notre approche et celle de la pire durée (pour une meilleure lisibilité, nous avons adopté une représentation par automate fini des tâches)

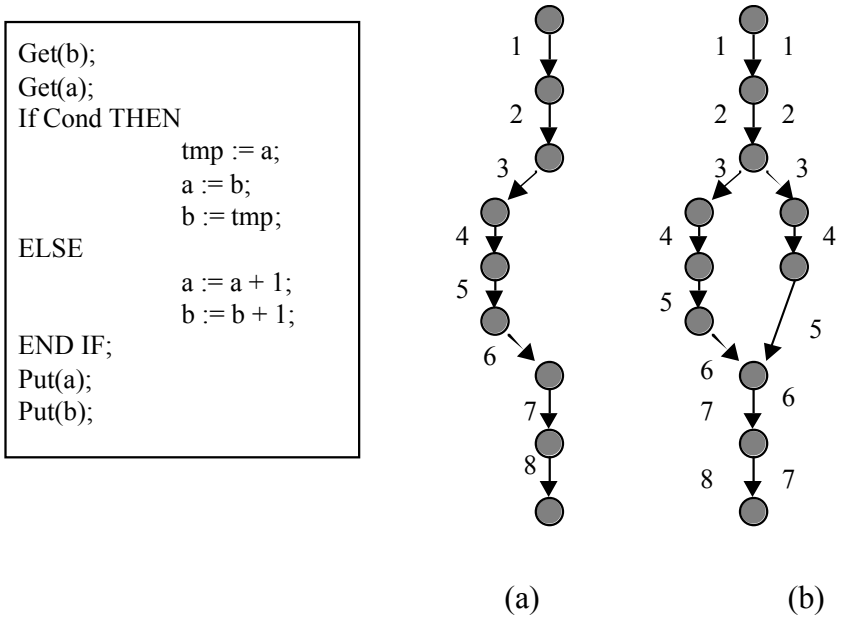


Figure 2. - Le Modèle temporel classique (a) : C = 8 et le modèle étendu (b) : D = {7, 8}.

Dans ce qui suit, nous considèrerons seulement les systèmes de tâches **à départs simultanés**, c'est à dire tels que $r_i = 0$ pour tout i. Il s'ensuit que les ordonnancements sont périodiques, de période P = le PPCM des périodes (méta-période). Nous réalisons donc l'étude d'ordonnançabilité dans la fenêtre temporelle [0..PPCM(périodes)[[LM80, GCG00b].

2.2. Arbre d'ordonnancement

La présence d'instructions conditionnelles n'affecte les techniques d'ordonnancement en ligne qu'au niveau de la validation de l'application et peuvent être utilisées directement : en effet, les décisions d'ordonnancement sont prises en fonction de l'état instantané du système, et des paramètres R et T des tâches prêtes. Par contre, les techniques d'ordonnancement hors ligne doivent être reprises et adaptées. Si nous utilisons le modèle classique, l'objectif d'une stratégie d'ordonnancement hors ligne est de construire une ou plusieurs séquences d'ordonnancement valides, destinées à être utilisées par le distributeur. Ceci ne peut pas être utilisé dans notre cas, car les différents choix dans les différentes conditionnelles induiront des comportements différents de l'application, qui ne pourront être décrits dans une unique séquence. Pour pallier ce problème (et donc éviter de recourir à l'optique du pire cas), nous définissons la notion **d'arbre d'ordonnancement** : c'est un arbre dont chaque branche correspond à une séquence d'ordonnancement de l'application obtenue en remplaçant les blocs conditionnels par l'une de leur branche (nous appellerons **éclaté** une telle application). Les nœuds de l'arbre sont simples s'ils correspondent à l'exécution d'une instruction impérative d'une tâche, et doubles s'ils correspondent à l'exécution d'un test. La figure 3 donne un exemple d'arbre d'ordonnancement, et la figure 4 illustre la notion d'éclaté. Notons que si l'application comporte c blocs conditionnels dans la fenêtre temporelle [0…PPCM(périodes)[, tous les arbres d'ordonnancement comporteront 2^c branches, ou ce qui est équivalent, l'application admettra 2^c éclatés.

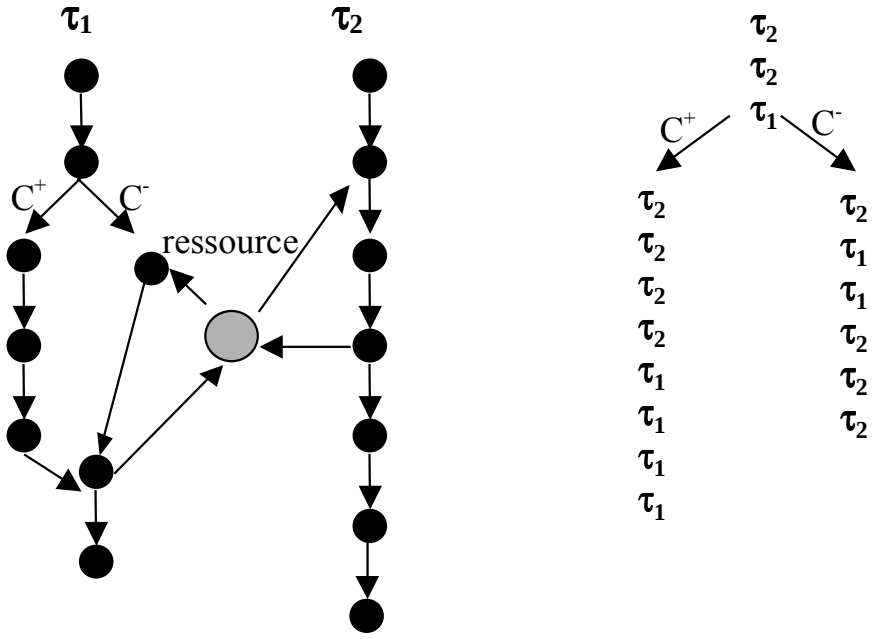


Figure 3. - Un arbre d'ordonnancement obtenu pour un système de 2 tâches. Il y a une instruction conditionnelle donc un seul nœud double. Nous avons explicité dans l'arbre les tests, pour les autres actions, nous avons seulement fait apparaître le nom de la tâche. Les deux branches de l'arbre correspondent aux séquences : $\tau_2\tau_2\tau_1C^+\tau_2\tau_2\tau_2\tau_2\tau_1\tau_1\tau_1\tau_1$ *et* $\tau_2\tau_2\tau_1C^-\tau_2\tau_1\tau_1\tau_2\tau_2\tau_2$.

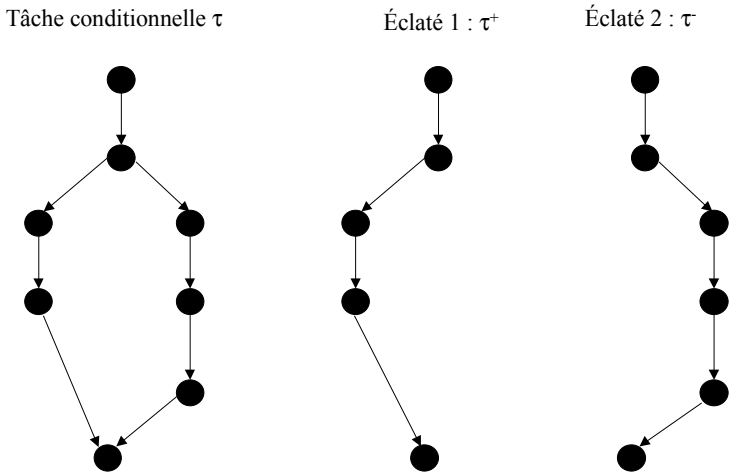
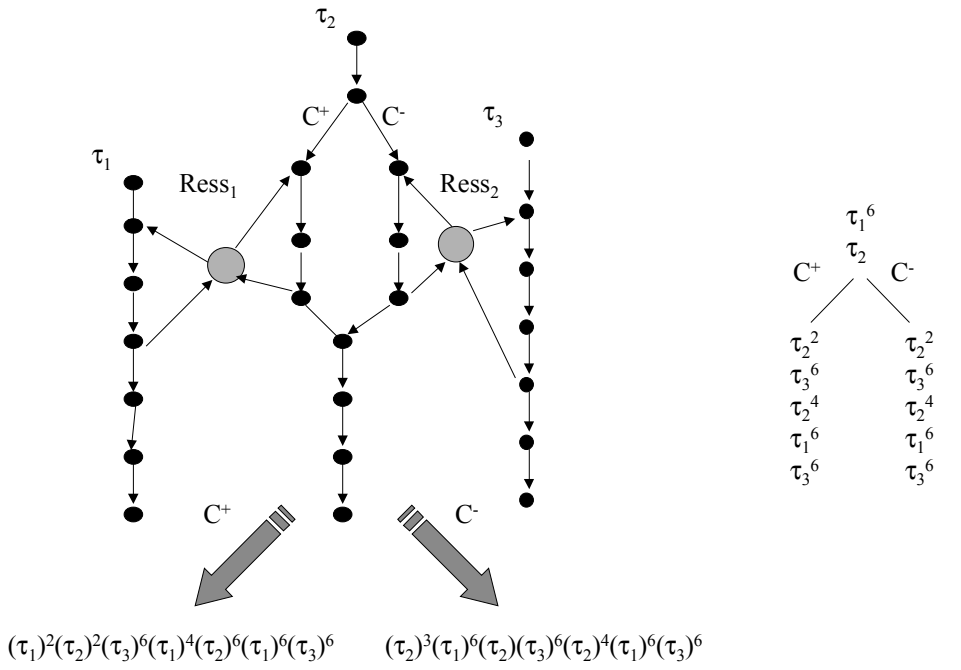


Figure 4. - La tâche conditionnelle τ admet deux éclatés τ^+ et τ^-.

Nous introduisons alors deux notions d'ordonnançabilité :

- Une application est dite **localement ordonnançable** si chacun de ses éclatés est ordonnançable, i.e. il existe pour chacun d'eux une séquence d'ordonnancement valide (voir figure 5).

- Une application est dite **globalement ordonnançable** s'il existe au moins un arbre d'ordonnancement valide, i.e. tel que toutes les échéances soient respectées le long de chacune de ses branches. L'application de la figure 5 est globalement ordonnançable.



(a) ordonnançabilité locale (b) ordonnançabilité globale

Figure 5. - Ordonnançabilités locale et globale de l'application τ_1 <$r_1=0$, $C_1=6$, $R_1=T_1=16$>, τ_2 <$r_2=0$, $C_2=8$, $R_2=T_2=32$> et τ_3 <$r_3=0$, $C_3=6$, $R_3=T_3=16$>.

L' ordonnançabilité globale suppose l'existence d'un séquenceur " omniscient ", capable de prendre des décisions valides quelque soient les futurs choix, alors que dans le cas de l'ordonnançabilité locale, les décisions d'ordonnancement sont prises en anticipant sur les choix à venir. Il résulte de cette remarque que les deux notions ne sont pas équivalentes, et que, pour ordonnancer une application contenant des blocs conditionnels, il ne suffit pas d'étudier individuellement chacun de ses éclatés. De manière plus précise :

- Une application globalement ordonnançable est localement ordonnançable. Ceci provient du fait que chacune des branches d'un arbre d'ordonnancement est une séquence valide pour l'un des éclatés.

Par contre, la réciproque est fausse. Considérons l'application τ_1<r_1=0, C_1=9, R_1=T_1=48>, τ_2 <r_2=0, C_2=5, R_2=T_2=8> et τ_3 <r_3=0, C_3=9, R_3= T_3=48> (voir figure 6). Elle est localement ordonnançable : on effectue l'étude sur la fenêtre temporelle [0..48[. Il y a 5 instances de τ_2, donc 5 embranchements, ce qui donne 2^5 = 32 éclatés. On peut vérifier que chacun d'eux est ordonnançable. Par exemple, la séquence $\tau_3^2\tau_2^5\tau_1\tau_2^5\tau_1^6\tau_2^5\tau_1^2\tau_3\tau_2^5\tau_2^5\tau_3^6\tau_2^5$ est une séquence valide pour l'application éclatée correspondant aux choix de l'alternative C^+ dans les 5 instances de τ_2, et la séquence $\tau_2^5\tau_1^5\tau_1\tau_2^5\tau_3\tau_1\tau_2^5\tau_3^6\tau_2^5\tau_3^2\tau_1\tau_2^5$ à une séquence valide si l'on effectue alternativement les choix C^+ et C^- dans les conditionnelles. Mais par contre, on ne peut pas l'ordonnancer globalement : en effet, si l'on se place à l'instant 8, on a exécuté une occurrence de τ_2 et 3 unités de temps ont été dédiés aux tâches τ_1 et τ_3. Donc au moins l'une des ressources est bloquée. Si l'on se place à l'instant 16, il a fallu exécuter une nouvelle occurrence de τ_2, et seules 3 unités de temps ont été dédiées aux deux autres tâches. Ceci est insuffisant pour que la ressource bloquée précédemment ait été libérée. Par suite, l'une des alternatives C+ ou C- de τ_2 conduira à un dépassement d'échéance.

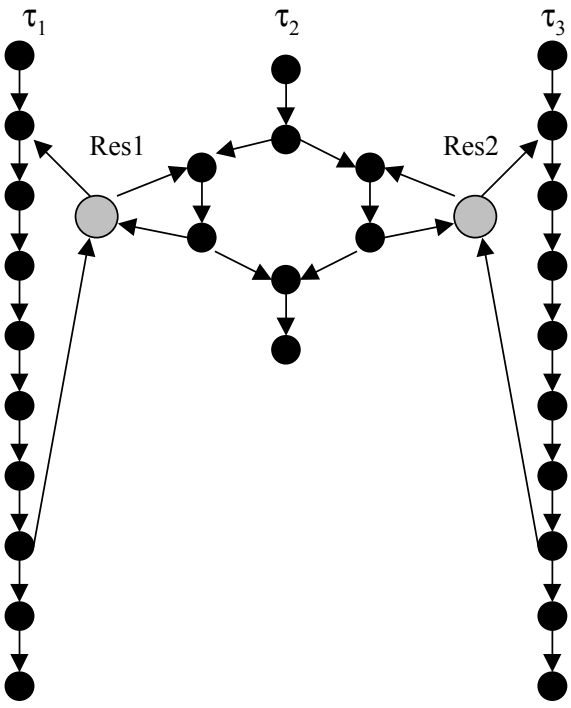


Figure 6. - L'ordonnançabilité locale n'implique par l'ordonnançabilité globale : l'application τ_1<r_1=0, C_1=9, R_1=T_1=48>, τ_2 <r_2=0, C_2=5, R_2=T_2=8> et τ_3 <r_3=0, C_3=9, R_3= T_3=48> est localement ordonnançable mais pas globalement.

Nous pouvons toutefois noter que les deux concepts coïncident si les blocs conditionnels ne comportent aucune primitive temps réel.

3. Modélisation de l'application

La méthodologie proposée pour étudier l'ordonnançablité globale d'une application temps réel est une adaptation de la méthodologie présentée dans [CGC96, GCG00]. 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 arborescences valides. Notre travail a consisté à adapter le modèle utilisé, puis à définir une procédure d'extraction des arbres d'ordonnancement.

3.1. Principe de la modélisation

Nous utilisons un réseau de Petri autonome, avec places colorées, fonctionnant sous la règle de tir maximal [Sta90], et muni d'un ensemble terminal [VVN81].

Le modèle adopté comporte deux parties (voir figure 7) : un réseau modélisant le système de tâches, et un réseau modélisant la structure temporelle.

3.1.1. Structure temporelle

Nous avons adopté un modèle discret du temps [Kop92, Foh94] : une horloge externe (RTC) comptabilise les ticks, une unité de temps correspondant à l'intervalle s'écoulant entre deux ticks successifs. Cette horloge est modélisée par une transition source RTC (voir figure 7). Chaque tir de la transition RTC représente un tick. Nous disposons ainsi d'une représentation logique du temps. Par ailleurs, à chaque tâche est associée une horloge locale composée d'une place Time permettant de comptabiliser le temps écoulé depuis la dernière activation de la tâche et d'une transition Clock qui tire toutes les périodes.

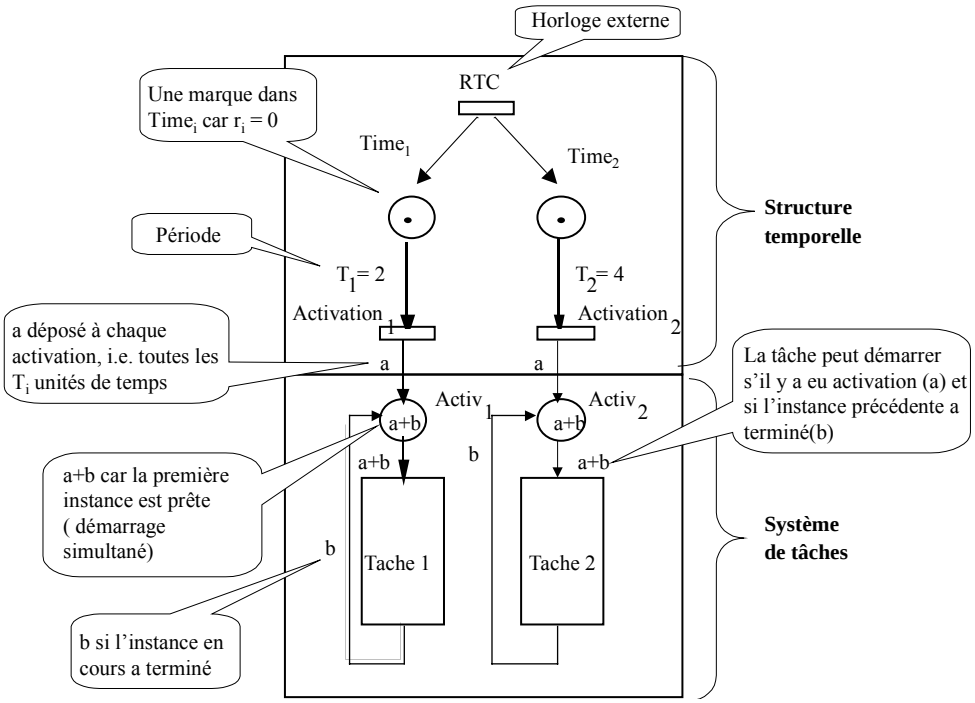


Figure 7. - Modélisation d'une application comportant 2 tâches de périodes respectives $T_1 = 2$ et $T_2 = 4$.

3.1.2. Le système de tâches

La modélisation de l'ensemble des tâches se fait de manière classique (voir figure 8). Notons simplement que chaque transition correspond à une action de durée une unité de temps, et que toutes ces transitons sont en concurrence pour l'obtention du processeur, modélisé par la place Processeur (en règle générale, pour des raisons de lisibilité, nous ne représenterons pas les arcs reliant cette place aux transitions du système de tâches).

3.1.3. Prise en compte des délais critiques

Les activations et terminaisons de la tâche sont modélisées par les jetons a et b. Une instance ne pourra démarrer que si la place Activation contient a + b. Les délais critiques sont alors pris en compte à l'aide de l'ensemble terminal suivant :

- $M(Time_i) = 1 \Rightarrow M(Activ_i) = \{a,b\}$ (cas où la tâche est à échéance sur requête, i.e. $T_i = R_i$) : au moment de la réactivation, l'instance précédente est terminée.

- $M(Time_i) = R_i+1 \Rightarrow M(Activ_i)=\{b\}$: une fois l'échéance passée, la tâche a terminé.

La figure 8 présente le réseau modélisant un système de 3 tâches se synchronisant et utilisant une ressource critique, auquel est adjointe la tâche oisive (voir section 3.2).

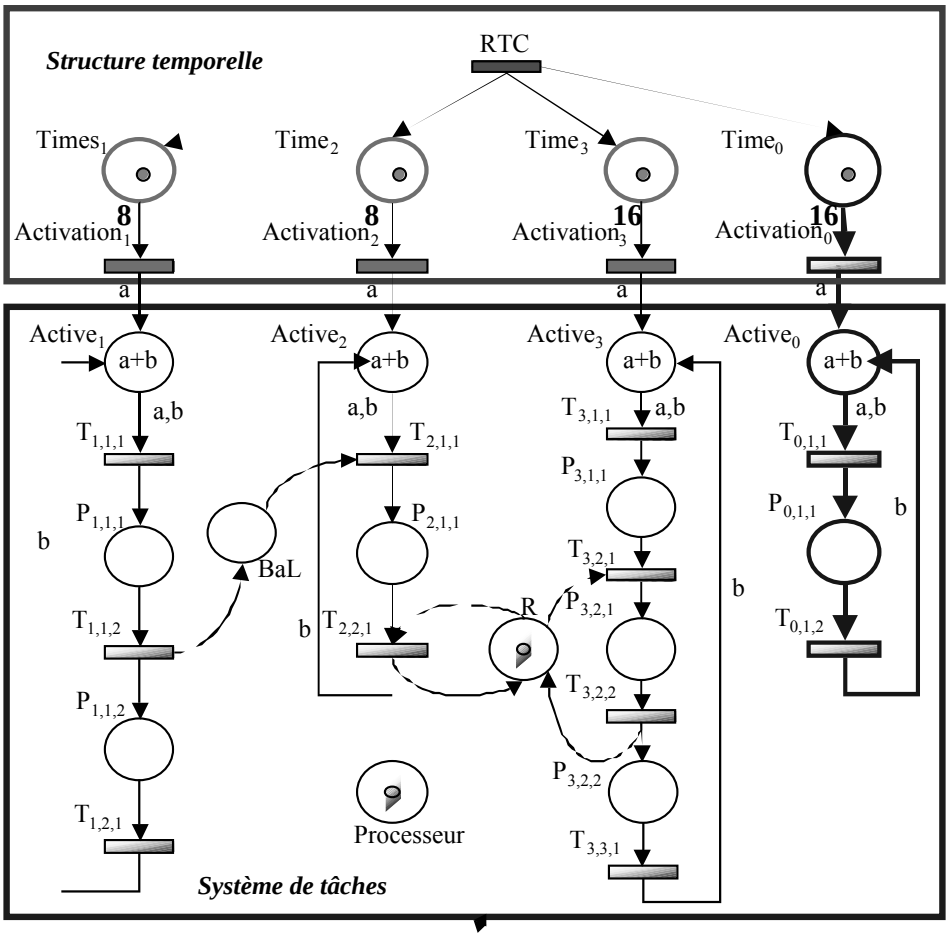


Figure 8 - Réseau modélisant le système τ_1 <r_1=0, C_1=3, R_1=T_1=8>, τ_2 <r_2=0, C_2=2, R_2=T_2=8> et τ_3 <r_3=0, C_3=4, R_3=14, T_3=16>, τ_0 <r_1=0, C_1=2, R_1=T_1=16>. La place Processeur est reliée en entrée et en sortie à toutes les transitions du système de tâche (non visualisé sur la figure). La place BaL est une boîte aux lettres permettant la modélisation de la communication, et la place R correspond à une ressource critique.

L'ensemble terminal est donné par :

$M(Time_1) = 1 \Rightarrow M(Active_1) = a + b$

$M(Time_2) = 1 \Rightarrow M(Active_2) = a + b$

$M(Time_3) > 14 \Rightarrow M(Active_i) = b$

$M(Time_0) = 1 \Rightarrow M(Active_0) = a + b$

3.2. Règle de fonctionnement et tâche oisive

3.2.1 Définition

A chaque système de tâches est adjointe une tâche τ_0 dite **tâche oisive** qui modélise l'inactivité du processeur (voir sur la figure 8).

Ses paramètres temporels sont : $r_0 = 0$, $C_0 = P(1 - U)$ où $U = \sum C_i / T_i$ est le **facteur de charge** de l'application et $P = PPCM(T_i)$ est la **méta-période**, $R_0 = T_0 = P$. Ainsi, nous travaillons avec un système de charge 100%.

Nous adoptons un fonctionnement sous la règle de tir maximal : à chaque instant, une transition du système de tâches (et une seule à cause de la concurrence liée au processeur) tire, ainsi que RTC et le cas échéant les transitions Activation. La présence de la tâche oisive garantit que l'on pourra obtenir aussi bien les séquences non conservatives que les conservatives puisque la tâche oisive peut être élue à tout instant. Cette tâche est donc un élément capital pour l'exhaustivité de la méthode proposée.

3.2.2. Modélisation

Dans le modèle initial, la tâche oisive est 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, ce qui garantit que τ_0 sera intégralement exécutée toutes les méta-périodes. Il est donc superflu de doter τ_0 d'un délai critique. L'usage des jetons a et b, ainsi que l'intégration de τ_0 dans l'ensemble terminal deviennent également superflus. La tâche oisive peut donc être modélisée à l'aide d'une seule place et d'une seule transition comme le montre la figure 9 : 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 B.

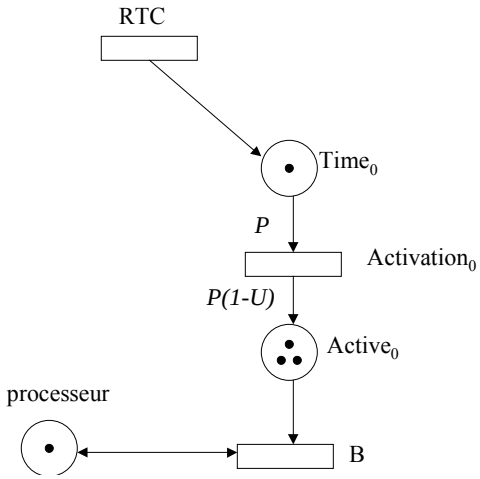


Figure 9. - Modélisation de la tâche oisive.

3.3. Prise en compte du modèle étendu

3.3.1. Les tâches à blocs conditionnels

La modélisation des branches conditionnelles se fait de manière classique (voir figure 10) par deux transitions concurrentes C^+ et C^- correspondant chacune à une alternative du test. Si la tâche se termine par un bloc conditionnel, chacune des branches doit se terminer par une transition qui dépose la marque b de terminaison dans la place Active. De cette façon, les ensembles terminaux définis dans le cas déterministe restent préservés.

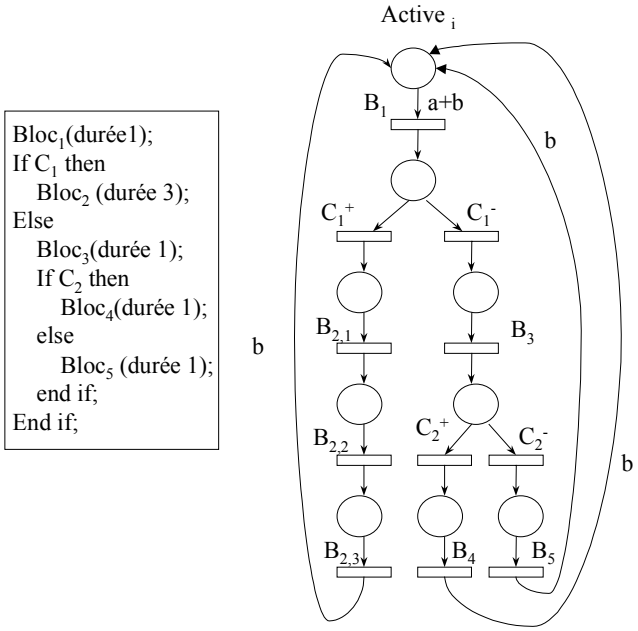


Figure 10. - Modélisation d'une tâche conditionnelle : les trois transitions terminales déposent b dans la place Active$_i$.

3.3.2. La tâche oisive

En présence de tâches à durées non fixes, 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* : on donne 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 τ_0 c'est à dire retirer des marques de la place Active$_0$, comme l'illustre la figure 11. L'inconvénient de cette approche est qu'il y a augmentation des retours en arrière 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 l'on désire faire une étude d'ordonnançabilité locale, et que l'on souhaite récupérer toutes les séquences valides pour chacun des éclatés.

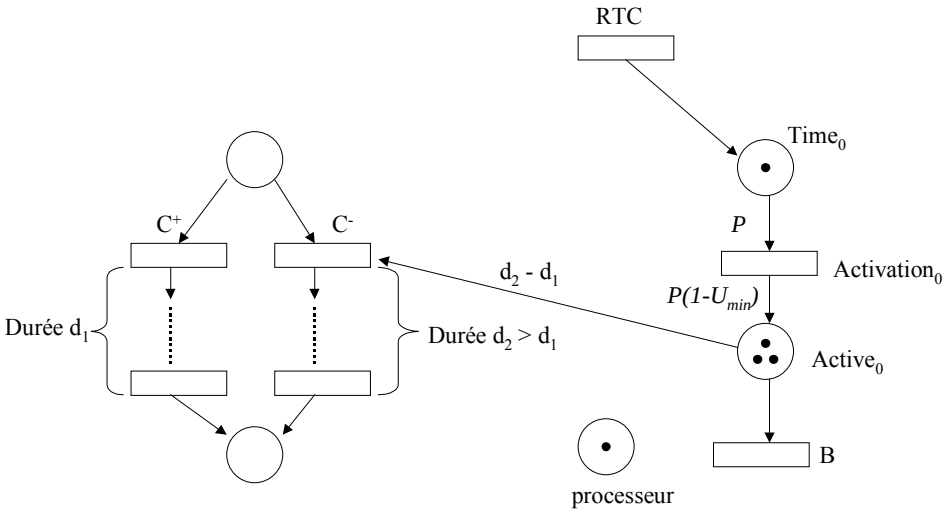


Figure 11. - Modélisation de la tâche oisive par la méthode de la plus longue durée.

- *Méthode de la plus courte durée* : cette fois, on utilise 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 12. 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.

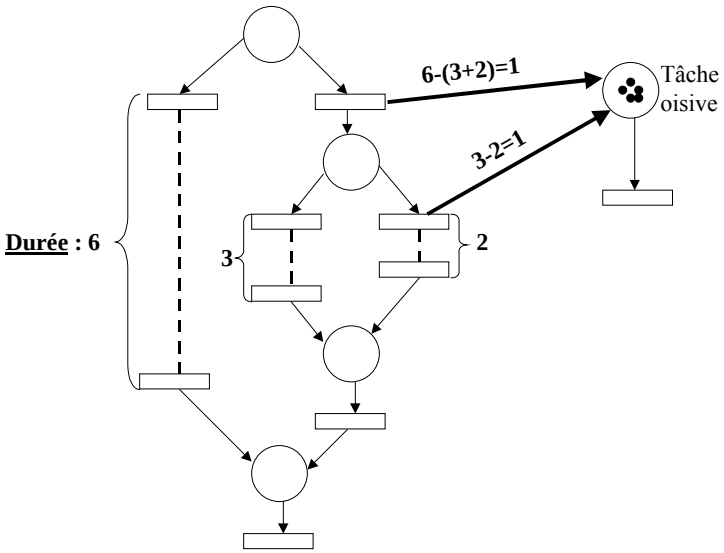


Figure 12. - Ajustement de la charge de la tâche oisive pour la méthode de la plus courte durée avec U = 0,75, P = 20 et donc durée (τ_0) = 5.

4. Analyse Hors-ligne

Nous venons de voir comment modéliser un système temps réel comportant des tâches conditionnelles à l'aide de réseau de Petri. Nous nous intéressons maintenant à l'exploitation de cette modélisation afin d'obtenir des arborescences d'ordonnancement.

A partir de ce réseau de Petri, nous construisons un graphe des marquages terminaux de profondeur P permettant d'analyser le système de tâches. Puis, il est nécessaire de supprimer

les états stériles, i.e. tels qu'il n'existe pas de séquences valides passant par ces états. Nous appelons **graphe d'accessibilité** le graphe ainsi réduit aux seuls états valides et non stériles.

L'ajout de tâches conditionnelles nécessite la redéfinition de la notion d'états stériles. Notre objectif, nous l'avons vu, est d'extraire les arbres d'ordonnançabilité globale. Nous devons dans ce but restreindre le graphe d'accessibilité afin d'éliminer les séquences orphelines, c'est à dire tel que quand on arrive en test , seule l'une des alternatives est envisageable. Ensuite nous devons définir un processus extraction d'arbres d'ordonnancement.

4.1. Le graphe des marquages

Nous savons que tout ordonnancement de tâches périodiques est cyclique, de période égale à la méta période P du système de tâches, dans un contexte de tâches à départs simultanés. Par conséquent, le graphe des marquages est fini, et l'état initial M_0 est un état d'accueil[1].

La construction du graphe 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 l'on souhaite 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).

- *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 si l'on utilisait une construction en largeur.

Par ailleurs, il faut supprimer les états stériles, qui sont ceux à partir desquels on ne peut pas atteindre l'état final (marquage M_f, à profondeur P). Un état non stérile est défini intuitivement par :

- *M_f est non stérile*

- *Un marquage terminal M de hauteur h est non stérile s'il existe un marquage terminal M' de hauteur h+1, non stérile et une transition t telle que M (t > M'.*

Notons que si l'on utilise l'approche de la plus longue durée par la tâche oisive, le graphe ainsi construit, contient toutes les séquences valides. Il permet donc d'étudier l'ordonnancement local.

L'étude de l'ordonnançabilité globale, par contre, nécessite une prise en compte conjointe des deux branches des instructions conditionnelles. La notion de nœud stérile doit donc être étendue : un nœud sera également stérile si à partir de ce nœud, on ne peut effectuer que des instructions conditionnelles, et si chaque fois que l'une des branches d'une alternative permet d'atteindre le marquage final, l'autre branche ne le permet pas.

Le définition d'un état non stérile devient donc :

- *M_f est non stérile*

- *Un marquage terminal de profondeur h est non stérile si : -soit il existe une transition t correspondant à une instruction non conditionnelle, et un marquage terminal non stérile M' tels que M (t > M', - soit il existe deux transition Cond+ et Cond-, correspondant*

[1] Un état d'accueil est un état accessible à partir de tout marquage accessible;

aux deux alternatives d'une test, et deux marquages terminaux non stériles M' et M'' tels que : M (Cond$^+$ > M' et M (Cond$^-$ > M''

Par conséquent, une fois le graphe d'accessibilité construit, il convient de l'épurer, de façon à le réduire à un graphe d'ordonnançabilité globale. L'algorithme d'épuration enlève du graphe d'accessibilité toutes les transitions conditionnelles dépareillées et les marquages devenant ainsi stériles. La figure 13 illustre le mécanisme d'épuration. On peut noter que la suppression d'un sommet n'implique pas nécessairement la disparition de tout son sous-arbre, car certains descendants de ce nœud peuvent avoir d'autres ancêtres non stériles (voir la figure 13).

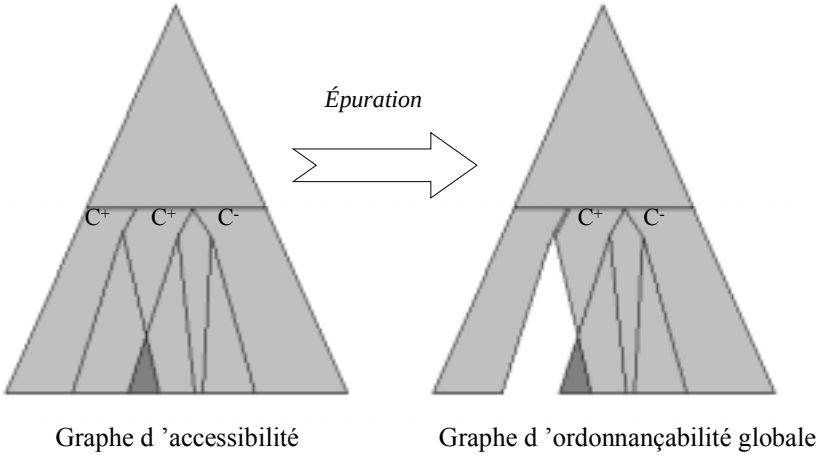


Figure 13 - Épuration du graphe d'ordonnançabilité locale.

Nous présentons ci-dessous un algorithme d'épuration du graphe d'ordonnançabilité globale. Nous nous plaçons dans une optique de construction complète du graphe.

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 nœud 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 *LienPère*, qui stocke tous les arcs qui pointent sur le nœud 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 nœuds 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
```

```
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
```

```
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
```

4.2. Extraction d'arborescence

Le graphe d'accessibilité que nous venons de construire regroupe toutes les arborescences d'ordonnancement global, partant du marquage initial M_0 de taille P. Nous devons maintenant en extraire individuellement un ou plusieurs arbres d'ordonnancement. Deux optiques peuvent être envisagées : si l'on souhaite simplement résoudre le problème de l'ordonnancement, il suffit d'extraire un unique arbre. Par contre, si l'on souhaite prendre en compte des critères qualitatifs tels qu'un temps de réponse optimal pour un sous ensemble de tâches, ou bien une exécution au plus tôt d'un sous ensemble de tâches, nous pouvons être amenés à construire d'abord l'ensemble de tous les arbres d'ordonnançabilité globale, puis à sélectionner parmi eux les arbres optimisant le critère considéré.

Du fait de la parité des transitions conditionnelles, l'opération d'extraction consiste à mettre en évidence des arbres binaires de racine M_0. Tous les nœuds possédant deux fils correspondent aux marquages où un test conditionnel est réalisé (voir figure 14).

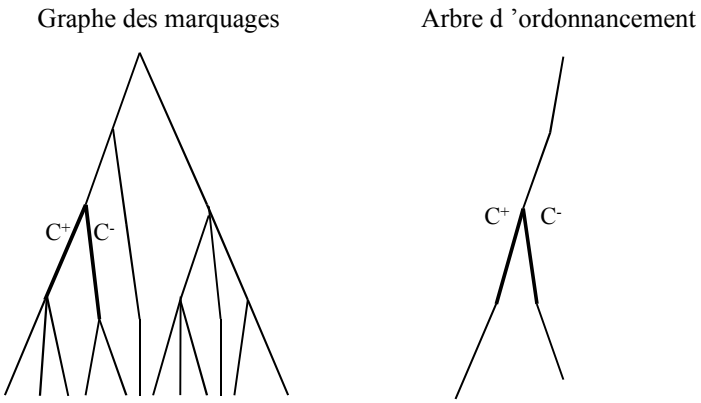


Figure 14. - Extraction d'un arbre d'ordonnancement

L'algorithme d'extraction que nous présentons permet l'extraction de l'ensemble de tous les arbres d'ordonnancements. Ceux-ci seront stockés dans une liste.

On part du marquage initial M_0. On choisit un arc issu de ce marquage, et on mémorise tous les autres arcs dans une **liste des autres choix (Liste_AC)**. Si l'arc choisi correspond à l'une des alternatives d'un test, l'arc correspondant est mémorisé non pas dans la liste des autres choix, mais empilé dans la **pile des alternatives (PileCond)**. On recommence ensuite l'opération à partir du marquage pointé par l'arc choisi, et ce jusqu'à atteindre la profondeur P. On dépile alors la pile des alternatives, et on construit ainsi, pour chaque alternatives, l'autre branche. Quand la pile des alternatives est vide, on a construit un arbre, on le mémorise dans un liste des arbres d'ordonnancement global. Ensuite, pour construire un nouvel arbre, on choisit le premier arc de la liste des autres choix, et on reprend la construction avec ce nouvel arc.

On utilise donc : - une pile *PileCond* qui contient un nœud du graphe des marquages (*Marq*) d'où part un test, un arc étiqueté par l'une alternative du test et un nœud de l'arbre d'ordonnançabilité (*Place*), - une liste *Liste_AC* qui contient un nœud du graphe des marquages, un arc et un nœud de l'arbre d'ordonnançabilité, - une fonction *Genre(Arc)* qui indique si un arc est un TEST, - une fonction *Complémentaire(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, un champ *end* pour pointer sur l'extrémité de l'arc.

```
PROCEDURE GenereAll (Graphe, Arbre, PileCond, Courant)
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)
               GenereAll(Graphe, Arbre, PileCond, Courant)
```

```
            FinSi
      Sinon
            Liste_AC :=Tous les arcs issus de Graphe
            Tant que Liste_AC non Vide faire
                  -- parcours de tous les arcs issus du nœud courant
                  Action :=Defile(Liste_AC)
                  Courant.FilsGauche := new Nœud (Action.Label, Null, Null)
                  Si (Genre(Action)=TEST) alors
                        -- l'arc courant est un TEST, on crée un fils droit,
                        -- on le stocke dans PileCond
                        Action2 :=Complémentaire(Liste_AC, Action)
                        Courant.FilsDroit :=new Nœud (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
                  GenereAll(Action.end, Arbre, PileCond,Courant.FilsGauche)
                  Courant.filsGauche :=Null
            FinTantque
      FinSi
Fin
```

La présence des tâches conditionnelles contribue toutefois considérablement à l'explosion combinatoire, il peut donc être préférable de choisir un algorithme d'extraction sélectif qui ne gardera qu'un seul arbre d'ordonnancement en choisissant à chaque étape de la construction de l'arbre quelle action effectuer. La figure 15 illustre les différentes étapes de cet algorithme.

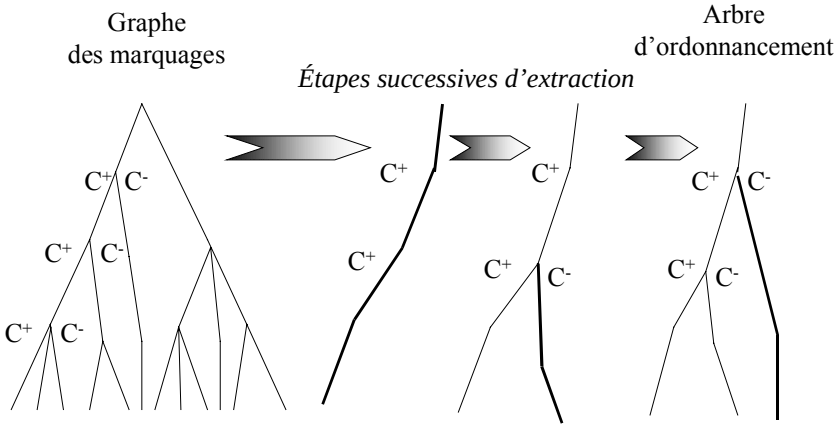


Figure 15 - Extraction d'un arbre d'ordonnancement.

L'algorithme ci-dessous correspond à l'extraction d'un unique arbre. Les notations sont les mêmes que précédemment.

```
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)
                GenereAll(Graphe, Arbre, PileCond, Courant)
        FinSi
    Sinon
        -- on peut apporter un critère particulier pour ne choisir q'un seul arc
        Action :=Choix d'un arc issu de l'état courant du Graphe
        Courant.FilsGauche :=new Nœud(Action.Label, Null, Null)
        Si (Genre(Action)=TEST) alors
                -- le nœud courant est un TEST, on crée un fils droit,
                -- on le stocke dans PileCond
                Action2 :=Complémentaire(Graphe Action)
                Courant.FilsDroit :=new Nœud (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
```

5. Conclusion

Nous avons proposé une méthodologie de prise en compte de tâches à durées variables soumises à des contraintes temporelles strictes. Ces variations de durées sont issues d'applications conditionnelles strictes que nous avons modélisées par Réseau de Petri en vue d'une analyse hors-ligne. Cette étude s'inscrit dans le cadre d'une extension des travaux proposé dans [CGC96]. Nous avons défini deux notions d'ordonnançabilité, issues directement des implications de l'intégration de tâches conditionnelles : l'ordonnançabilité locale et l'ordonnançabilité globale, et nous avons montré qu'il n'y a pas équivalence entre les deux.

Par ailleurs, nous avons revu la notion de tâche oisive, car la durée de cette tâche est étroitement liée aux fluctuations de durées des tâches de l'application. Nous avons donc proposé deux modélisations permettant au processeur de disposer d'un nombre adéquat de temps creux, l'une des modélisation étant adaptée à l'étude de l'ordonnançabilité locale et l'autre à celle de l'ordonnançabilité globale.

Nous nous sommes ensuite intéressés à l'exploitation du graphe des marquages obtenu à partir du réseau de Petri modélisant le système, de façon à extraire tous arbres d'ordonnancement valides, ces arbres correspondant à l'adaptation de la notion de séquence dans le cas où apparaissent des branches conditionnelles.

L'étape suivante de cette étude consistera à étudier les techniques de choix des arbres vis à vis des divers critères qualitatifs comme par exemple la minimisation du temps de réponse.

Par ailleurs, d'autres extensions sont envisagées comme la prise en compte de tâches sporadiques par l'intermédiaire d'activations conditionnelles. Ce qui nous amènera à étudier plus profondément la construction du graphe des marquages et des arbres d'ordonnancement, afin de proposer un service des tâches sporadiques adapté.

6. Bibliographie

[Bab96] J.P. BABAU, "Étude du comportement temporel des applications temps réel à contraintes strcites basées sur une analysse d'ordonnançabilité", Thèse de Doctorat, Université de Poitiers, Juillet 1996

[Bak91] T.P.BAKER, " Stack-based Scheduling of Realtime Processes ", *The Journal of Real-time systems*, n°3, p. 67-99, Kluwer Academic Publishers, 1991

[CGC96] A.CHOQUET-GENIET, D.GENIET, F.COTTET "Exhaustive Computation of the scheduled Task Execution Sequences of a Hard Real-time Application", *4th symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Uppsala, Sweden, LNCS n° 1135, p. 246-262, Springer Verlag, September 1996

[GCG00] E.GROLLEAU, A.CHOQUET-GENIET, " Off-line computation of real time schedule by means of Petri nets ", *Discrete events systems*, eds. R.Boel et G. Stremersch, p. 309-316, Kluwer academic Publishers, Wodes 2000, Gent, Août 2000

[GCG00b] E.GROLLEAU, A.CHOQUET-GENIET, " Cyclicité des ordonnancements des systèmes de tâches périodiques différées ", *RTS'2000*, p. 216-229, Paris Mars 2000

[CHB79] R.H. CAMPBELL - K.H. HURTON, G.G. BELFORD, "Simulations of fault-tolerant real-time deadlin mechanisms ", Proceedings of FCTS'9, p. 95-101, Janvier 1979

[Che91] H. CHETTO, M. CHETTO, "An adaptative scheduling algorithm for fault-tolerant real-time systems", Software Engineering Journal, p. 179-186, Mai 1991

[CL88] J.Y. CHUNG, J.W.S LIU, « Algorithms for scheduling periodics jobs to minimize average error », 9th IEEE RTSS, p.142-152, Décembre 1988

[CL90] M.CHEN, K.LIN, " Dynamic priority ceilings : a concurrency protocol for real-time systems ", *The Journal of Real-Time Systems* 2 (n°4), p. 325-346, 1990

[Foh94] G.FOHLER, "Flexibility in statically scheduled hard real-time systems", Thèse de l'université de Wien, Autriche, Avril 1994

[Kai82] C.KAISER, " Exclusion mutuelle et ordonnancement par priorité ", *Technique et Science Informatiques*, vol 1, no 1, p. 59-69, 1982

[Kop92] H. KOPETZ, "Sparse time versus dense time in distributed real-time systems", 12th Int. Conf. On Distributed Computing Systems, p. 460-467, Japon, Juin 1992

[LL73] C.L. LIU, J.W. LAYLAND, "Scheduling algorithms for multiprogramming in a hard real-time environment", Journal of the ACM, 20 (1), pp 46-61, 1973

[LM80] J.Y.T.LEUNG, M.L.MERILL, " A note on preemptive scheduling of periodic real-time tasks ", *Information Processing Letters* 11(3), p. 115-118, 1980

[LSD89] J. LEHOCZKKY, L.SHA, Y. DING "The rate monotonic scheduling algorithm : exact characterization and average case behavior" Proceedings of the 10th IEEE real-time systems symposium, pp 166-171, 1989

[MD78] A.K. MOK, M.L. DERTOUZOS, "Multi processor scheduling in a hard real-time environment", 7th Texas conference on computer Systems, 1978

[SRL90] L.SHA, R.RAJKUMAR, J.LEHOCKZY, " Priority inheritance protocols : an approach to real time synchronisation ", *IEEE transaction computers*, vol 39, N°9, p. 1175-1185, 1990

[SSR98] J.A. STANKOVIC, M. SPURI , K. RAMAMRITHAM, G. C. BUTTAZZO, "Deadline scheduling for real-time systems", Kluwer Academic Publishers, ISBN 0 7923 8269 2, 1998

[Sta90] P.STARKE, " Some properties of timed Petri nets under the earliest firing rule ", *Advances in Petri nets*, LNCS n° 424, p. 418-432, 1990

[VVN81] R.VALK, G.VIDAL-NAQUET, " Petri nets and regular languages ", *Journal of Computer and System Sciences*, vol. 23, n° 3, p.299-325, Academic press, 1981