Intégration d'Éléments Sémantiques dans l'Analyse d'Ordonnançabilité des Applications Temps-Réel

Christian Fotsing Takoutsi

Laboratoire d'Informatique Scientifique et Industrielle
École Nationale Supérieure de Mécanique et d'Aérotechnique
École Doctorale Sciences et Ingénierie pour l'Information
Téléport 2- 1 Avenue Clément Ader- Bp 40109, 86961, Futuroscope
Téléphone: 05. 49. 49. 83. 36 Fax: 05. 49. 49. 80. 64

THÈSE

pour l'obtention du grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE

(Diplôme National - Arrêté du 07 août 2006)

École Doctorale : Sciences et Ingénierie pour l'Information

Secteur de Recherche : Informatique et Applications

Présentée et soutenue par

Christian FOTSING TAKOUTSI

Intégration d'Éléments Sémantiques dans l'Analyse d'Ordonnançabilité des Applications Temps-Réel

Directeurs : Pr. Annie CHOQUET-GENIET, Pr. Guy VIDAL-NAQUET

Soutenue le 20 Février 2012

Devant la Commission d'Examen :

JURY :

Président :	Pr. Emmanuel GROLLEAU, ENSMA
Rapporteurs :	Pr. Franck POMMEREAU, Université d'Évry
	Pr. Frank SINGHOFF, Université de Brest
Examinateurs :	Dr. Christophe AUSSAGUÈS, CEA - LIST
	Pr. Annie CHOQUET-GENIET, Université de Poitiers, ENSMA
	Pr. Guy VIDAL-NAQUET, Université de Paris-Sud, SUPELEC

À mes parents
Odette Kougoum et Jean Takoutsi

Remerciements

"*La reconnaissance silencieuse ne sert à personne*"
Gladys Bronwyn Stern

En m'excusant par avance pour tous ceux dont les noms n'apparaitront pas sur cette page, faute d'espace, je tiens à remercier :

L'éternel *DIEU* tout puissant pour la vie qu'il me donne, et la force qu'il m'a insufflée jusqu'à nos jours. Je ne le remercierai jamais assez pour tous ses bienfaits.

Le Conseil Général de la région Poitou Charentes pour la bourse qui m'a été attribuée afin que je puisse accomplir sereinement mes recherches.

Le professeur *Guy Pierra* qui m'a accueilli et intégré au LISI. J'espère avoir été à la hauteur de la confiance qu'il m'a accordée.

Annie Geniet, mon directeur de thèse, pour la formation à la recherche, la qualité de son encadrement, son soutien tant sur le plan académique que familial, sa disponibilité, sa rigueur scientifique et enfin sa promptitude à corriger mes rapports. Sans elle, ce rapport ne serait surement pas comme il est dans l'état actuel.

Guy Vidal-Naquet, mon co-directeur de thèse, pour son regard critique, et la pertinence de ses remarques tout au long de ma thèse, mais aussi pour l'ouverture (à travers un choix sélectif de mes conférences et séminaires) et sa disponibilité.

Franck Pommereau et *Frank Singhoff* pour l'honneur qu'ils m'ont fait en acceptant d'être les rapporteurs de ma thèse, ainsi que *Christophe Aussaguès* et *Emmanuel Grolleau* pour avoir accepté de faire partir de mon jury de thèse.

Mes formateurs de l'Université de Yaoundé 1, de l'Institut Africain d'Informatique, de l'Université de Poitiers et de l'École Nationale Supérieure de Mécanique et d'Aérotechnique : cette thèse est le résultat de votre travail.

Mes collègues du LISI (mon laboratoire). L'ambiance que vous avez créée au laboratoire prédispose à fournir d'excellents résultats.

Mes anciens collègues de l'IAI (*promotion no limits*), mes amis de Poitiers (*club un zéro*), les familles *Fotsing*, *Médérer*, *Missè*, *Piembeck*, *Rancher*, *Takoutsi*, *Tzazefack*, *Zonga Bana*, ... pour tous nos échanges et votre soutien.

François Missè Ngoh et *Bernadette Marie Assen Aliguen*. Vous m'avez toujours considéré comme votre fils. J'espère qu'aujourd'hui vous êtes fiers de moi.

Ma famille et ma belle-famille, particulièrement : *Éric*, *Féli*, *Mu*, *Taroc the Lion* (tu l'es), *Francko*, *Hermann*, *Baggio*, *Ireine*, *Élodie*, *Alice*, *Edwige* (ma grand-mère), *Guillaine*, *Madeleine*, *Yves-Eric*, *Léo*, (ma) *Lili*, *Manu*. Merci pour votre affection, votre soutien et votre compréhension durant cette thèse qui m'a un peu éloigné.

Aline Babette Missè Missè, mon épouse, ma confidente, mon amie, mais aussi *Glorya Angelye Fotsing*, ma fille, ma grande. Vous êtes mes reconforts et remparts, les femmes qui me supportent tous les soirs. Vous me rendez chaque jour meilleur.

Table des matières

Table des figures

Chapitre 1

Introduction générale

"*L'observation scientifique est toujours une observation polémique*"
Gaston Bachelard

Sommaire

Nous introduisons dans ce chapitre la notion de systèmes temps-réel, puis définissons la problèmatique de notre sujet de thèse : comment prendre en compte la sémantique des tests conditionnels pendant l'analyse d'ordonnançabilité des systèmes. Nous concluons en présentant brièvement la méthodologie utilisée pour résoudre le problème, et en donnant les grands axes de notre contribution.

1.1 Les systèmes temps-réel

1.1.1 Définitions

Une première définition tirée de [Stankovic 1988] 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.*
On classe généralement en trois catégories [Elloy 1997] [Cottet 2005] les systèmes informatiques :

1. les systèmes transformationnels qui utilisent des données fournies à l'initialisation par l'utilisateur. Ces données, leurs traitements et l'obtention du résultat n'ont aucune contrainte de temps.

On trouve par exemple dans cette catégorie les logiciels scientifiques de calcul, les logiciels de gestion de base de données ;

2. les systèmes interactifs dans lesquels les évènements sont produits par interaction avec l'environnement sous différentes formes (clavier, fichier, réseaux, souris ...). Le temps n'intervient pas dans ces systèmes, sauf pour apporter un aspect confort de travail ou qualité de service.
   Les systèmes transactionnels et les outils bureautiques sont des exemples de systèmes interactifs ;
3. les systèmes réactifs, où les évènements à traiter sont issus du monde extérieur, mais pour lesquels l'aspect temps a une place importante, soit en terme de temps de réaction, soit d'une échéance à respecter.
   Ce sont des systèmes informatiques de contrôle commande.

Cette classification des systèmes informatiques est schématisée figure 1.1.

Types de systèmes	Aspects considérés
Systèmes transformationnels	Elaboration des résultats
Systèmes interactifs	Relations entre les entités du programme
Systèmes réactifs	Réaction à un évènement

Figure 1.1 – Classification des systèmes informatiques

Dans ce rapport de thèse, nous nous intéressons aux systèmes réactifs, pour lesquels le temps de réaction a une importance : ce sont les **systèmes temps-réel**.
Nous considérons donc des systèmes temps-réel constitués de tâches qui sont des réponses à des évènements [Kopetz 1995].
Ces évènements peuvent être liés à une horloge, dans ce cas on parle de systèmes dirigés par le temps (**Time Triggered Systems**), ou pas, et dans ce cas on parle de systèmes dirigés par les évènements (**Event Triggered Systems**).
Dans cette thèse, les systèmes considérés sont **dirigés par le temps**, les dates d'activation des tâches sont ainsi connues à l'avance.
De façon générale, un système de contrôle de procédé est un système informatique temps-réel. Celui-ci reçoit des informations sur le procédé contrôlé à travers des capteurs, puis traite ces informations, et en fonction des résultats obtenus, envoie des commandes au procédé à travers des actionneurs, cela dans un temps donné.
Les informations envoyées par les capteurs peuvent être de type interruptions (information tout ou rien) ou de type mesures (information continue).
Les commandes envoyées par les actionneurs peuvent être de type modification d'une

trajectoire ou affichage sur un écran.
Ces commandes doivent parvenir au procédé dans un temps donné, qui dépend de la dynamique du procédé, afin de ne pas être obsolètes.
Ceci illustre la différence entre un système temps-réel et un système rapide.
Le principe du fonctionnement des applications temps-réel de contrôle-commande est donné figure 1.2.

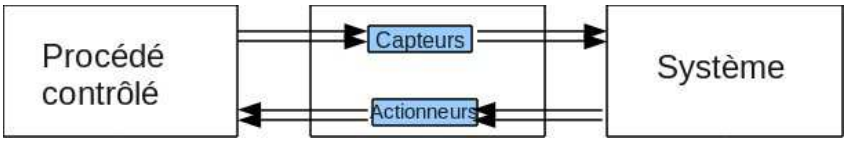


Figure 1.2 – Fonctionnement global d'un système de contrôle-commande

Les systèmes temps-réel rencontrés sont souvent des systèmes de contrôle-commande.
D'où la définition du CNRS [CNRS 1988] :
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.
Nous nous intéressons dans le cadre de notre thèse à la validation des contraintes temporelles des applications temps-réel.
Pour suivre le modèle usuel, qui déconseille pour des raisons de structuration et de maintenance d'utiliser une approche monolithique (utilisation d'une seule tâche pour décrire l'application), les applications sont représentées comme un ensemble de tâches en interaction.
Les valider nécessite en particulier de démontrer que chaque tâche peut s'exécuter dans le délai qui lui est imparti. C'est le *problème de l'ordonnancement*.
Deux aspects de la validation des systèmes peuvent être considérés suivant l'importance accordée au respect des contraintes temporelles :

1. si on ne tolère aucun dépassement de contraintes, il s'agit du **temps-réel dur**.
   Dans ce cas, le non respect des contraintes peut conduire à des situations critiques, voire catastrophiques.
   C'est le cas dans des situations de pilotage automatique, de système de surveillance de centrale nucléaire ... ;
2. si par contre certains dépassements peuvent être tolérés, il s'agit du **temps-réel mou**.
   Le système peut s'accommoder de dépassements de contraintes temporelles dans certaines limites, au delà desquelles il devient inutilisable.
   C'est le cas pour la visioconférence, les jeux en réseaux ....

Dans notre travail, nous ne nous intéressons pas aux conséquences d'un non respect des échéances.
Notre étude consistera à vérifier que les échéances sont respectées.

Les systèmes manipulés sont des **systèmes temps-réel**[1] **durs**.

1.1.2 Exemples

Nous citons dans ce paragraphe quelques systèmes temps-réel issus de plusieurs domaines. Certains sont tirés de [Cottet 2005] :

1. le robot de production.
   Un robot qui réalise une activité spécifique (assemblage, tri ...) sur une chaine de production doit effectuer son travail dans des délais fixés par la cadence de fabrication. S'il agit trop tôt ou trop tard, l'objet traité sera endommagé, et cela peut conduire à des conséquence financières ou humaines graves.
   C'est par exemple le cas pour l'oubli par un robot d'un ou de plusieurs rivets lors de l'assemblage d'un avion (temps-réel dur) ;
2. la téléphonie mobile.
   Le système doit pouvoir respecter ses contraintes fonctionnelles (transmettre et recevoir les signaux de la parole) tout en rendant les contraintes temporelles imperceptibles à l'utilisateur pour avoir une bonne qualité de service (temps-réel mou) ;
3. le système airbag des véhicules.
   Le véhicule est équipé de capteurs qui détectent les chocs. En cas de choc violent, les actionneurs gonflent l'airbag, améliorant ainsi la sécurité du conducteur (temps-réel dur).

Afin de pouvoir étudier les systèmes temps-réel et les contraintes temporelles, [Liu 1973] propose une représentation formelle.
C'est l'objet de la section 1.1.3.

1.1.3 Modélisation classique

Les systèmes temps-réel étant modélisés comme un ensemble de tâches interactives, nous commençons par définir la notion de tâche.

1.1.3.1 Notion de tâche

Une **tâche** est une unité active de l'application. Il s'agit d'un ensemble d'instructions séquentielles.
C'est un ensemble d'instructions exécutées sur apparition d'un même évènement ou d'un même ensemble d'évènements.
Dans ce travail, les tâches que nous manipulons sont constituées :

1. d'instructions de calculs internes (blocs indépendants d'une durée donnée correspondant au temps d'exécution du bloc) ;
2. d'instructions de communication (prise/libération de ressources, envoi/réception de messages).

[1] Dans cette thèse, nous parlerons indifféremment de système temps-réel ou d'application temps-réel

Nous nous intéresserons aux **instructions conditionnelles** qui peuvent concerner les instructions de type 1 et 2. Elles seront modélisées par des **blocs conditionnels**.

Le cycle de vie d'une tâche est décrit par la figure 1.3.

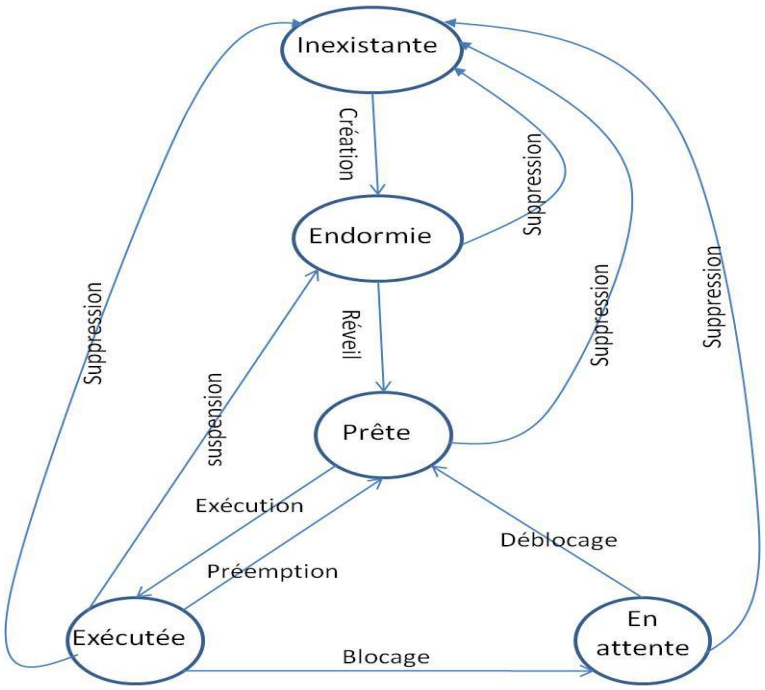


Figure 1.3 – Graphe des états possibles d'une tâche temps-réel

Une tâche est initialement **créée**, ainsi elle passe dans l'état **endormie**.
Après son réveil, elle passe dans un état dit **prête**.
C'est dans cet état qu'elle requiert un processeur (dans notre cas, la tâche est une entité séquentielle non parallélisable).
Cette tâche peut donc se voir allouer un processeur, afin d'être **exécutée**.
De l'état exécuté, une tâche peut être **pré-emptée** par une autre, ou bien se mettre en attente d'un message, d'une date, d'un événement ou de l'accès à une ressource.
Lorsqu'elle est en attente, une tâche passe dans l'état **prête** lorsque l'événement, la date, ou le message (la ressource) est arrivé (libre).
Si un ensemble de tâches est tel que chacune est en attente d'un événement (tel que la réception d'un message, la libération d'une ressource), que seule une autre tâche de l'ensemble peut engendrer, on dit qu'elles sont en situation d'**inter-blocage**.
Dans un contexte plus général, une tâche peut être **supprimée**.
Enfin, une tâche qui s'exécute peut se mettre en attente pendant un certain temps ou jusqu'à une certaine date : on parle de **suspension**.

Suivant la cyclicité de leur exécution, il existe plusieurs types de tâches temps-réel :

1. les tâches **périodiques** [Liu 1973] : ce sont celles pour lesquelles les évènements traités se répètent à intervalles réguliers.
   Cet intervalle est la **période** ;
2. lorsque l'évènement ne se répète pas à intervalles réguliers, la tâche n'est pas périodique.
   On distingue dans ce cas :
   (a) les tâches **sporadiques** [Kim 1980] [Mok 1983] : ce sont des tâches dont on ne connait pas à priori la période, mais dont on connait l'intervalle minimal séparant deux requêtes successives ;
   (b) les tâches **sporadiquement périodiques** [Audsley 1993] : elles sont activées par des événements qui arrivent à un certain rythme pour un certain nombre de fois, puis une longue pause est effectuée et le processus reprend de nouveau.
   Elles forment ainsi une suite de tâches qui est déclenchée de manière périodique.
   Les tâches sporadiquement périodiques ont deux périodes :
      i. la **période interne** : c'est le temps minimum séparant deux activations successives à l'intérieur de la suite de tâches ;
      ii. la **période externe** : c'est le temps séparant deux activations successives de la suite de tâches ;
   (c) les tâches **apériodiques** [Sprunt 1989] [Strosnider 1995] : ce sont les tâches pour lesquelles le délai minimal entre deux occurrences est inconnu.

Dans le cadre de cette thèse, nous supposons que les tâches manipulées sont périodiques.
Nous avons fait ce choix parce qu'il s'agit du cas le plus fréquent. En effet, un grand nombre de systèmes temps-réel est constitué de tâches périodiques, et les résultats les plus simples et les plus généraux ont été élaborés pour ce modèle de tâches. De plus, c'est un modèle bien adapté pour les systèmes temps-réel critiques.

Les tâches peuvent aussi se classer suivant les échanges qu'elles entretiennent.
On trouve ainsi les systèmes avec des tâches :

1. **dépendantes** : lorsque les tâches partagent des ressources ou échangent des messages.
   Un cas particulier de tâches dépendantes est celui des tâches avec **contraintes de précédences** (c'est le cas lorsque une(plusieurs) tâche(s) doit(doivent) être terminée(s) avant qu'une(que plusieurs) tâche(s) ne se réveille(nt)) ;
2. **indépendantes** : dans le cas où il n' y a aucun échange de message, ni aucune ressource en commun.

1.1.3.2 Modélisation temporelle classique d'une tâche

Dans les modélisations classiques utilisées dans la littérature, les blocs conditionnels sont encapsulés par des séquences constituées de blocs indépendants et de primitives temps-réel [Pushner 1989] [Wilhelm 2008].
Une tâche $T_i =< r_i, C_i, D_i, P_i > (i \in \mathbb{N}^*)$ est ainsi modélisée par quatre paramètres temporels [Liu 1973] :

1. la **date de réveil** r_i correspond à la date de première activation de la tâche ;
2. la **pire durée d'exécution** C_i (on parle aussi de WCET pour Worst Case Execution Time) est le temps le plus long d'utilisation du processeur par la tâche, c'est à dire le temps maximum qu'il lui faut pour s'exécuter entièrement si elle dispose en permanence du processeur ;
3. le **délai critique** D_i correspond au temps alloué à la tâche pour se terminer après chacune de ses activations ;
4. la **période** P_i est la durée entre deux activations successives de la tâche.
   La tâche est donc réactivée à chaque instant $r_i + k \times P_i$, où k est un entier naturel.
   Chacune de ces réactivations constitue une **instance** de la tâche.

Nous supposons pour notre étude que $C_i \leq D_i \leq P_i$. Les tâches considérées sont donc non **ré-entrantes**.
Par définition, une tâche est **ré-entrante** quand une de ses instances peut être activée, alors que la précédente n'a pas terminé son exécution.
Dans le cas où $D_i = P_i$, on dit que la tâche est à **échéance sur requête**.

Nous pouvons utiliser les dates de réveil pour classer les tâches :

1. lorsque les dates de réveil sont identiques, on dit que les tâches sont à **départs simultanés** ;
2. dans le cas contraire, on dit que les tâches sont à **départs différés**.

Nous donnons figure 1.4 une représentation graphique des paramètres temporels d'une tâche périodique.
Dans le modèle présenté figure 1.4, les paramètres r_i, D_i et P_i sont fournis par le concepteur.
Quant au paramètre C_i, il existe plusieurs méthodes de calcul, permettant de l'obtenir [Puschner 1997] [Bernat 2002] [Puaut 2007] [Wilhelm 2008].
On considère en général trois grandes techniques de calcul du WCET :

1. les techniques basées sur le calcul dynamique.
   Elles consistent à mesurer le temps d'exécution du programme considéré sur un système réel ou un simulateur.
   Ces mesures doivent être réalisées pour tous les jeux d'entrées possibles, ou alors il faut être capable de définir un jeu d'entrées dont on est certain qu'il conduit au temps d'exécution le plus long ;

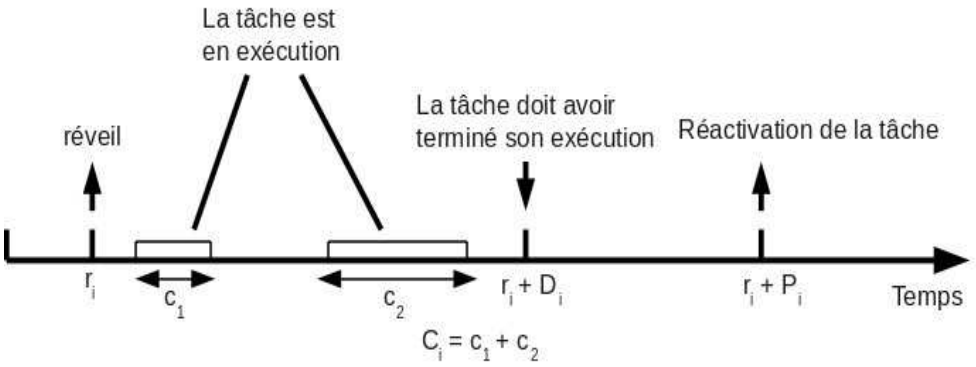


Figure 1.4 – Modélisation classique d'une tâche T_i

2. les techniques basées sur le calcul statique.
   Elles sont fondées sur une analyse statique du programme dans le but de s'affranchir des jeux d'entrées ;
3. les techniques mixtes.
   Elles combinent les deux approches afin de fournir un WCET moins pessimiste.

D'un côté les estimations obtenues par les méthodes statiques sont sûres, mais pessimistes.
De l'autre, celles obtenues par les méthodes dynamiques sont moins surévaluées, mais pas toujours très fiables.
Et enfin, les méthodes mixtes permettent à ce jour (au moment de la rédaction de cette thèse) d'obtenir un WCET moins pessimiste que celui qui serait calculé de manière statique, mais il n'est connu que de manière probabiliste.
Dans cette thèse, nous ne nous intéressons pas au calcul de C_i.
Nous le supposons connu pour les portions de code de tâches dont nous aurons besoin.

Certains paramètres importants pour l'étude des systèmes se déduisent des précédents :

1. la **charge processeur** d'une tâche qui est définie par $U_i = \frac{C_i}{P_i}$.
   Intuitivement, il s'agit du taux d'occupation du(des) processeur(s) par la tâche T_i ;
2. le **temps de réponse** d'une tâche qui correspond à la durée entre l'activation d'une tâche et sa date de fin d'exécution ;
3. le **facteur d'utilisation** (ou charge processeur ou facteur de charge) d'un système $S = \{T_1 \dots T_n\}$ de n tâches $T_1 \dots T_n$, $n \in \mathbb{N}^*$ qui est défini par $U = \sum_{i=1}^{i=n} U_i$.
   Il définit le taux global d'occupation du(des) processeur(s) par l'ensemble des tâches du système ;

4. la **fenêtre temporelle minimale** qui correspond à un intervalle sur lequel la validation du système garantit l'absence définitive de faute temporelle, c'est à dire une violation de contrainte temporelle.
   Des études sur la cyclicité ont permis de calculer cette durée dans le cas d'un système monoprocesseur [Leung 1980] [Choquet-Geniet 2004].
   Notons que dans le cas d'un système S de n tâches périodiques de période respective $P_1 \dots P_n$, à départs simultanés, cette fenêtre est **l'hyperpériode** et vaut $P = PPCM(P_1, \dots, P_n)$ où $PPCM(P_1, \dots, P_n)$ désigne le plus petit multiple commun de toutes les périodes.

1.1.4 Validation des systèmes temps-réel

Valider une application temps-réel nécessite en particulier la réalisation d'une validation temporelle.
Celle-ci impose que l'on vérifie que chaque instance de chaque tâche la constituant respecte son échéance. C'est à dire que l'on doit vérifier qu'elle s'exécute dans le temps qui lui est imparti.
On dit dans ce cas que le système est **ordonnançable**.
L'étude qui consiste à effectuer cette validation s'appelle **l'analyse d'ordonnançabilité**.
Le **problème de l'ordonnancement** [Choquet-Geniet 2003] consiste à définir une politique d'attribution du processeur qui assure qu'aucune faute temporelle ne sera commise, c'est à dire qu'aucune instance de tâche ne terminera après son échéance.
Le **problème de la validation** [Choquet-Geniet 2003] de l'application consiste à déterminer s'il existe une solution au problème de l'ordonnancement.
Deux approches peuvent être envisagées :

1. les **approches en-ligne** [Liu 1973] [Dertouzos 1974] [Leung 1982] [Dertouzos 1989] [Choquet-Geniet 2003] où une politique d'ordonnancement est implémentée dans l'ordonnanceur, et l'algorithme choisi à chaque instant la(les) tâche(s) à attribuer au(x) processeur(s) ;
2. les **approches hors-ligne** [Xu 1992] [Zamorano 1997] [Grolleau 2002] [Largeteau 2002] [Choquet-Geniet 2003] qui consistent à fournir à l'ordonnanceur une séquence d'ordonnancement (ou un ordonnancement) pré-calculée. Dans ce cas, l'ordonnanceur n'a plus qu'un *simple* rôle de séquenceur.

De façon générale, si une application temps-réel est ordonnançable, alors son facteur de charge U vérifie $U \leq m$ (où m est le nombre de processeurs) [Buttazo 1997].
Par conséquent, pour un système temps-réel donné, dès que $U > m$, on peut conclure qu'il n'est pas valide. Les fautes temporelles ne peuvent pas être évitées.
Par ailleurs, il suffit de trouver un ordonnancement valide du système de tâches qui constituent une application temps-réel pour conclure que cette application est ordonnançable.

Nous donnons figure 1.5 un exemple d'ordonnancement valide de trois tâches indépendantes $T_1 = < 0, 2, 4, 4 >$, $T_2 = < 0, 2, 6, 6 >$ et $T_3 = < 0, 2, 12, 12 >$ sur un

processeur.

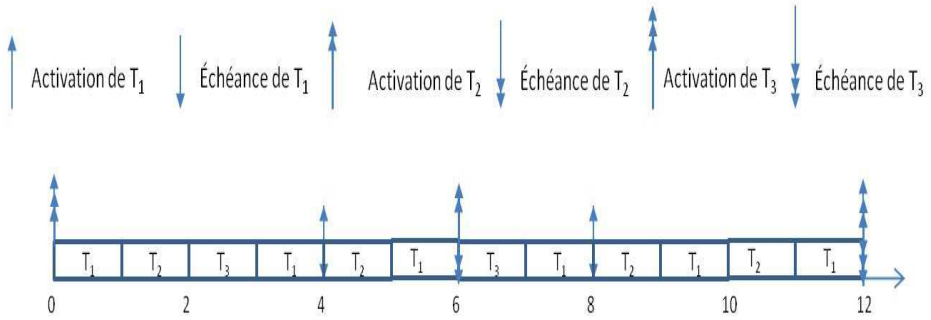


Figure 1.5 – Un ordonnancement valide de trois tâches indépendantes T_1, T_2 et T_3 sur un processeur

La tâche T_1 s'exécute pendant 2 unités de temps (une à la première unité et l'autre à la quatrième unité sur sa première période entre les instants 0 et 4), cela avant 4 unités de temps (qui est le délai critique), et est réactivée toutes les 4 unités de temps (la période). ...
Le lecteur peut, en suivant le même raisonnement pour les autres tâches, remarquer que l'ordonnancement est valide.

Dans la section 1.2 nous précisons les problèmes que nous abordons dans cette thèse, et le contexte de l'étude.

1.2 La problématique

Nous nous proposons dans cette thèse *d'intégrer des éléments sémantiques dans l'analyse d'ordonnançabilité des applications temps-réel.*
Concrètement, la validation actuelle des applications temps-réel repose sur la modélisation classique du système (voir section 1.1.3.2). Or, comme nous l'avons souligné, le paramètre C_i de chaque tâche T_i est obtenu en étudiant le corps de la tâche.
Dans ce calcul, nous voulons être sûr que, dans tous les cas de figures, les contraintes temporelles sont vérifiées.
Par contre, l'effectivité des différents cas de figure n'est pas prise en compte. Cela veut dire qu'on considère la plus longue durée d'exécution, afin d'éviter que la charge processeur du système soit sous-évaluée.
Notre objectif est d'étudier la pertinence du remplacement des blocs conditionnels par des séquences déterministes lors de la phase de modélisation de l'application, et ensuite de l'analyse d'ordonnançabilité : la question est de savoir si les conclusions d'ordonnançabilité sont exactes, ou bien si elles sont parfois trop pessimistes.
De façon plus précise, ne risque-t-on pas de perdre certains ordonnancements, et donc de conclure à la non ordonnançabilité d'une application, pourtant ordonnan-

çable dans les faits ?
De plus, surtout en présence de primitives temps-réel, la simulation reste un outil de validation très utile.
L'autre question est de pouvoir simuler le comportement effectif de l'application, y compris en présence de conditionnelles.
Les simulations doivent pouvoir rendre compte des différents comportements de l'application, ce que ne permet pas l'encapsulation systématique des conditionnelles.
Les problèmes rencontrés par l'encapsulation sont de différentes natures :

1. tout d'abord, l'encapsulation peut conduire à une sur-utilisation des ressources, alors qu'une description plus fine pourrait permettre de limiter l'utilisation de certaines d'entre elles à certains comportements.
   Donc avec l'encapsulation, des retards inutiles dus à une mobilisation supposée de certaines ressources pourront apparaître, amenant à des dépassements d'échéances en fait évitables ;
2. ensuite, la durée de chacun des blocs *encapsulants* est toujours celle du comportement le plus long de la conditionnelle qu'il englobe.
   Le comportement résultant, en terme de durée, n'est pertinent que si les différentes conditionnelles sont indépendantes entre elles.
   Une description plus détaillée permettra de tenir compte des liens existants entre les choix de comportements des différentes conditionnelles ;
3. enfin, un certain nombre de comportements dépendent de l'état de l'environnement au moment où la tâche s'active.
   Si plusieurs tâches agissent en fonction des mêmes paramètres, les choix de comportements de chacune d'elles seront corrélés.
   Ne pas tenir compte de ces corrélations peut conduire à envisager des scénarios qui ne se produiront jamais en pratique, car mettant en jeu des conclusions incohérentes quant à l'état de l'environnement.
   Si ces scénarios conduisent à des fautes temporelles, les conclusions d'ordonnançabilité ne seront pas nécessairement pertinentes.

Nos objectifs sont donc :

1. d'illustrer les problèmes posés par la considération des seuls modèles linéaires ;
2. de proposer une modélisation alternative, plus fine, tout d'abord des tâches, puis de l'ordonnancement ;
3. de proposer enfin une méthode adaptée d'analyse d'ordonnançabilité.

Nous commençons par délimiter le domaine d'étude, en situant le contexte de notre travail.

1.2.1 Le contexte de l'étude

Les applications considérées sont composées de tâches qui :

1. sont périodiques ;
2. sont à départs différés ou simultanés ;
3. peuvent être pré-emptées, partagent des ressources et s'échangent des messages ;
4. sont non ré-entrantes ;
5. peuvent comporter des instructions conditionnelles.

Ce travail s'inscrit enfin dans un contexte de système temps-réel en environnement monoprocesseur. Les tâches sont donc exécutées sur un unique processeur.

1.2.2 Hypothèses liées à l'étude

1.2.2.1 Sur les tests conditionnels

La particularité de notre étude est qu'elle porte sur des tâches possédant des instructions conditionnelles qui sont prises en compte de façon explicite.
Nous prenons en compte également les éventuelles relations qui peuvent exister entre les différents choix de comportements dans les différentes tâches.
Nous nous intéressons aux relations découlant de l'évaluation de tests portant sur l'examen de valeurs fournies par l'extérieur, par exemple des tests de type seuillage ($x < 5$) ou identification ($contact = on$), portant sur des paramètres en entrée des tâches (**Input**).
Ces paramètres ne peuvent donc pas être modifiés par la tâche elle même.
Ce sont par exemple des valeurs issues de l'observation de l'environnement, et qui ont été transmises au même moment aux différentes tâches.
Les variables de test peuvent provenir par exemple des tâches qui lisent des informations sur un capteur et les transmettent.
Par exemple, dans un système simplifié de contrôle de la température dans une pièce, une tâche T_i a une variable température qui reçoit l'information sur la valeur de la température lue, et la transmet à une tâche T_j qui selon la valeur, active une alarme, et à une tâche T_k qui selon la valeur, modifie le débit d'air froid.
L'étude de l'aspect sémantique de l'application concerne la prise en compte effective de ces tests pendant l'analyse d'ordonnançabilité.
Deux cas nous intéressent :

1. les relations **intra-tâches**.
   À l'intérieur d'une tâche, nous considérerons les corrélations entre les comportements choisis dans des conditionnelles successives, portant sur une même valeur.
   Nous considérons pour illustration l'exemple de la figure 1.6 où une tâche *alarme* prend en entrée la quantité de méthane dans l'air, et suivant cette valeur, déclenche une alarme de niveau 1 ou de niveau 2.
   Le corps de cette tâche est constitué d'une succession de blocs conditionnels, avec des décisions différentes.
   Nous pouvons remarquer que, pour une quantité de méthane connue en entrée

```
Tâche alarme
Input quantité_méthane;
{
        ..................................................;
        ..................................................;
        si (quantité_méthane < seuil1) alors
                ne rien faire;
        sinon
                déclencher l'alarme niveau 1;
        finsi;
        si (quantité_méthane < seuil2) alors
                déclencher l'alarme niveau 1;
        sinon
                déclencher l'alarme niveau 2;
        finsi;
        ..................................................;
        ..................................................;
        fin.
}
```

Figure 1.6 – Illustration de la relation intra-tâche

de la tâche, et pour $seuil_1 < seuil_2$, si la quantité de méthane est plus petite que $seuil_1$, le traitement de l'information $quantité\_méthane \geq seuil_2$ est inutile.
Le choix d'un comportement au niveau d'un test est corrélé au choix du comportement au niveau du(des) test(s) précédent(s).
La gestion des relations intra-tâches consiste à en tenir compte.
Remarquons pour finir que les relations intra-tâches sont similaires aux chemins impossibles de [Puaut 2007] [Wilhelm 2008] ;

2. les relations **inter-tâches**.
   Si plusieurs tâches examinent le même paramètre pour décider de leur comportement, leurs comportements respectifs seront liés.
   Pour l'illustrer, nous considérons l'exemple de la figure 1.7.

```
Tâche calcul
Input quantité_méthane
{
        ..................................;
        ..................................;
        si (quantité_méthane < seuil1) alors
                action1;
        sinon
                action2;
        finsi;
        ..................................;
        ..................................;
        fin.
}
```

```
Tâche commande
Input quantité_méthane
{
        ..................................;
        ..................................;
        si (quantité_méthane < seuil2) alors
                action3;
        sinon
                action4;
        finsi;
        ..................................;
        ..................................;
        fin.
}
```

Figure 1.7 – Illustration de la relation inter-tâches

Les tâches *calcul* et *commande* prennent en entrée la quantité de méthane. En considérant que $seuil_1 < seuil_2$, il n'est pas possible d'exécuter, pour la même quantité de méthane, les actions $action_1$ et $action_4$.
C'est ce type de relations entre les résultats des tests des tâches que nous étudierons.

D'autres tests peuvent exister (des tests qui ne portent pas sur des variables **Input** par exemple). Ils ne sont pas pris en compte par l'étude des corrélations intra-tâches ou inter-tâches. Par contre, bien entendu, ils seront considérés lors de la modélisation complète des tâches.

1.2.2.2 Sur la cohérence structurelle de l'application

Afin d'assurer la cohérence structurelle des applications étudiées, nous posons certaines hypothèses sur l'utilisation des primitives temps-réel.
Ce sont des hypothèses sur les instructions de prise et de libération de ressources et les instructions d'envoi et de réception de messages pour éviter d'une part les pertes de messages, et d'autre part des situations d'inter-blocage :

1. si une ressource R est prise dans une branche conditionnelle, elle est libérée dans cette branche conditionnelle, ou bien dans toutes les éventuelles branches conditionnelles qui suivent ;
2. si une ressource est prise à l'extérieur d'une branche conditionnelle, soit elle est libérée hors de tous les blocs conditionnels, soit elle est libérée dans toutes les branches d'un bloc conditionnel qui la suit ;
3. l'envoi d'un message M n'est pas bloquant, mais sa réception l'est ;
4. tous les messages émis sont effectivement reçus ;
5. tous les messages attendus sont effectivement émis.

Notons que pour éviter les pertes de messages, les systèmes considérés doivent en plus vérifier l'hypothèse :
si i (j) désigne le numéro d'une tâche émettrice (réceptrice), n_i (n_j) le nombre de messages émis (reçus) par la tâche T_i (T_j), on a : $\sum_i \frac{n_i}{P_i} = \sum_j \frac{n_j}{P_j}$.
Cette hypothèse garantit que la fréquence d'envoi des messages est la même que la fréquence de réception dans le système étudié.
De plus, nous supposons que :

1. les envois de messages ne se font pas en début de tâches : la tâche envoie un message après avoir exécuté une action ;
2. de même, les réceptions de messages ne se font pas à la fin des tâches : un message reçu transmet une information qui sera utilisée par la tâche réceptrice.

Nous supposons enfin que les durées des primitives temps-réel sont incluses dans les durées des blocs d'exécution adjacents. Elles sont donc dans la suite supposées de durée nulle.

Cette hypothèse facilite la définition formelle des primitives temps-réel, et leur utilisation pour la suite du rapport.

Nous pouvons remarquer que les hypothèses considérées dans cette étude (section 1.2.2.1 et 1.2.2.2) sont réalistes et rencontrées dans la plupart des systèmes temps-réel de contrôle-commande qui lisent des informations sur des capteurs à chaque période et effectuent des traitements sur les données acquises.

Les hypothèses et le cadre du travail étant posés, nous allons maintenant présenter un exemple que nous utiliserons dans tout le document, pour illustrer les problèmes et les solutions proposées.

Exemple 1.2.2.1 : Le système des essuie-glaces d'un véhicule
Nous considérons un système simplifié de contrôle-commande des essuie-glaces d'un véhicule.
Le rôle de ce système est d'activer automatiquement les essuie-glaces en cas de pluie, et de les désactiver par temps sec.
Un capteur de pluie posé sur le pare-brise permet d'avoir la quantité d'eau qui y tombe.
Ce système de contrôle-commande se retrouve de plus en plus dans les véhicules récents (voir figure 1.8).

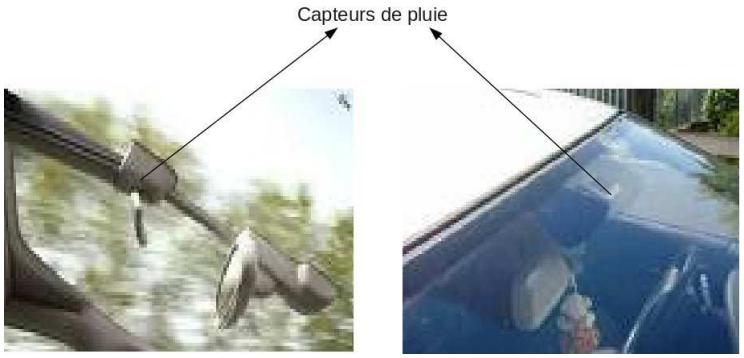


Figure 1.8 – Quelques capteurs de pluie dans la pratique

Ce système de contrôle est un système temps-réel critique.
En effet, pour limiter le risque d'accident lié à une mauvaise visibilité, le système doit déclencher les essuie-glaces dans un délai très court (celui qui a été fixé par le concepteur).
Le système des essuie-glaces est spécifié par les règles suivantes :

1. la vitesse du véhicule détermine la vitesse de balayage des essuie-glaces ;
2. un capteur posé sur le pare-brise permet d'obtenir la quantité d'eau déposée ;
3. un calculateur utilise ces données et la position du *comodo* (Stop/1/2/3) pour contrôler les essuie-glaces.

1.3 Objectifs et intérêts du travail

Notre objectif dans cette thèse est tout d'abord de montrer que lorsque les tâches des systèmes temps-réel contiennent des instructions conditionnelles, les approches classiques (linéaires) ne fournissent pas toujours des conclusions d'ordonnançabilité exactes. Elles sont parfois plus pessimistes que nécessaire.
Ensuite, nous proposerons une méthodologie adéquate de prise en compte de ces instructions conditionnelles (une modélisation effective des systèmes temps-réel), afin d'avoir des conclusions d'ordonnançabilité plus réalistes, et de produire un ordonnancement qui prend en compte les comportements des tâches de l'application.

La figure 1.9 résume la problématique de notre étude.

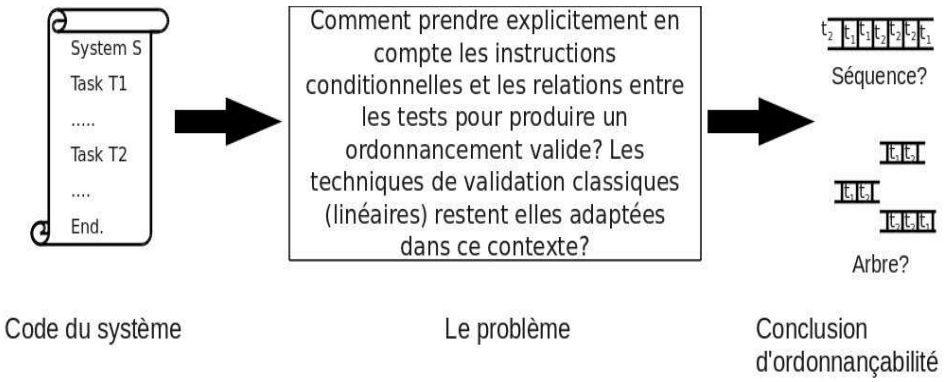


Figure 1.9 – Problématique de l'étude

Notons pour finir que l'analyse que nous proposons présente un double intérêt :

1. un **intérêt scientifique**.
   Nous effectuons une analyse réaliste des systèmes : nous avons une vue réelle de l'application à implémenter, et nous produisons des résultats plus fins.
   Ce qui évite de rater des ordonnancements ;
2. un **intérêt économique**.
   Puisque nous ne surévaluons pas les pires durées d'exécution, nous ne surdimensionnons pas le matériel (processeurs ...).
   Ce qui peut engendrer un gain tant financier qu'énergétique.

1.4 Contribution : méthodologie globale

Dans notre contexte, il n'existe pas de stratégies optimales en ligne. Cette notion est précisée section 3.2.1 du chapitre 3. Pour cette raison, et parce que nous souhaitons étudier les comportements de l'application, nous avons opté pour une approche hors ligne à base de modèles.
L'objectif est d'utiliser cette approche afin de sélectionner un ordonnancement de

l'application, et de proposer une simulation de son comportement qui intègre les comportements conditionnels.
Lorsque les tâches sont linéaires, mais dépendantes, une méthode orientée modèle à base de réseaux de Petri a été développée [Grolleau 1999] [Grolleau 2002].
Par ailleurs, modéliser des comportements conditionnels à l'aide des réseaux de Petri est naturel. Nous avons donc choisi d'étendre la méthode de [Grolleau 1999] [Pailler 2006], en intégrant la modélisation détaillée des tâches, ainsi que la modélisation des corrélations entre certains comportements.
Ensuite, l'analyse d'ordonnançabilité repose sur la construction du graphe de marquages et son analyse.
De la même façon qu'une tâche comportant des instructions conditionnelles est représentée par un arbre, alors qu'une tâche sans instruction conditionnelle admet une représentation linéaire, un ordonnancement représenté de façon linéaire en l'absence de tâches à instructions conditionnelles sera représenté par un arbre si certaines tâches comportent des instructions conditionnelles.
Ces arbres seront extraits du graphe des marquages, et intègreront la sémantique comportementale de l'application, c'est à dire qu'ils tiendront compte des corrélations entre les choix des différentes branches conditionnelles.
Nous les appellerons **arbres d'ordonnancement** par opposition aux séquences d'ordonnancement.

Nous illustrons figure 1.10 notre approche globale de résolution.

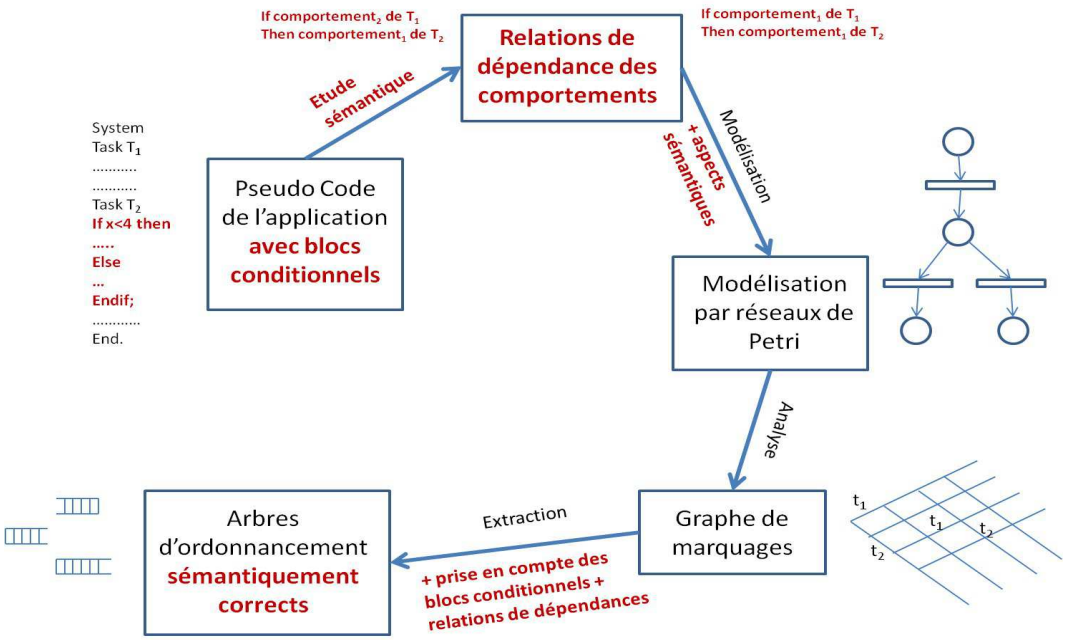


Figure 1.10 – Méthodologie globale de résolution

À partir du pseudo-code d'une application temps-réel donnée, la recherche d'un ordonnancement valide consiste à en proposer une modélisation par réseaux de Petri, qui tient compte de façon explicite des relations de dépendances comportementales entre les résultats des différents tests conditionnels de l'application.
Il restera ensuite à extraire des arbres d'ordonnancement valides du graphe des marquages construit à partir du modèle.
Notons que cette dernière étape d'extraction des arbres à partir du graphe des marquages n'a pas été traitée dans cette thèse, faute de temps.

D'autres techniques de génération de séquences auraient pu être utilisées.
Mais elles sont soit de complexité très élevées (automates finis) [Largeteau 2002], soit de puissance de modélisation inférieure aux réseaux de Petri (modèles géométriques discrets) [Largeteau 2005].
Le choix de l'approche réseaux de Petri basée modèle est à notre connaissance celui qui permet de prendre en compte une large classe de systèmes.
Cette approche est certes elle aussi exponentielle, mais en utilisant de bonnes heuristiques, sa complexité peut être réduite.

1.5 Plan et contenu de la présentation

Le présent manuscrit est divisé en deux grandes parties :

1. la premiére partie fait l'état de l'art de l'existant et revient sur l'intérêt de notre étude et nos motivations.
   Cette partie comporte deux chapitres :
   (a) le premier chapitre fait une synthèse des travaux réalisés qui intègrent les instructions conditionnelles dans le modèle de tâche.
   Il en ressort qu'en général, les auteurs qui ont travaillé sur ce thème n'ont pas étudié les relations entre les tests conditionnels des tâches.
   De plus, dans la plupart des cas, les auteurs fournissent un diagnostic de l'application (c'est à dire permettent d'avoir un critère d'ordonnancement satisfaisant de l'application), mais ne donnent pas une vue effective du comportement du système avant son implémentation ;
   (b) le deuxième chapitre présente les travaux de [Grolleau 1999] et [Pailler 2006].
   Il s'agit d'une approche de validation hors-ligne des systèmes temps-réel basée sur les réseaux de Petri.
   Le modèle de réseaux de Petri utilisé pour modéliser les systèmes temps-réel y est présenté, ainsi que la technique de génération des séquences d'ordonnancement.
   Une section est consacrée aux travaux de [Pailler 2006] pour la prise en compte des conditionnelles.
   Notons que son travail ne traite pas des relations sémantiques entre les

différents tests des tâches de l'application (ce que nous avons appelé relations intra-tâches et inter-tâches), et que la notion d'arbre d'ordonnancement n'est pas complètement définie ;

2. la deuxième partie constitue notre contribution à la prise en compte explicite des instructions conditionnelles et des relations entre les tests conditionnels des tâches pendant l'analyse d'ordonnançabilité.
   Cette partie comporte cinq chapitres :
   (a) dans le premier chapitre, nous étendons le modèle classique de tâche, afin qu'il prenne en compte les instructions conditionnelles.
   Nous présentons trois aspects de modélisation d'une tâche : le modèle structurel, le modèle d'exécution et le modèle temporel.
   Ces trois aspects permettent de modéliser complètement les tâches d'une application et de prendre en compte les relations intra-tâches.
   Deux approches découlent du modèle d'exécution : une basée sur les chemins, et l'autre basée sur les arbres d'exécution des tâches.
   Ces deux approches seront utilisées pour la modélisation des interactions entre les tâches et la génération des arbres d'ordonnancement.
   Une étude comparative sera faite au chapitre 8 ;
   (b) le second chapitre est consacré à la modélisation des interactions entre les tâches.
   Nous commençons par définir formellement la notion de relation d'incompatibilité entre différentes tâches suivant le modèle considéré (arbre d'exécution ou chemin), puis nous proposons une définition formelle des contraintes structurelles qui portent sur les primitives temps-réel.
   Nous concluons ce chapitre en donnant une définition formelle de la tâche oisive.
   La tâche oisive est la tâche qui occupe le processeur quand aucune des tâches du système ne s'exécutent.
   Pour une étude complète du système, elle doit être modélisée, en vue de la génération des arbres d'ordonnancement ;
   (c) dans le troisième chapitre, nous donnons une définition formelle des arbres d'ordonnancement, et proposons un algorithme d'obtention de ces arbres.
   Il s'agit de générer de façon *brute* tous les arbres d'ordonnancement.
   Cet algorithme a une complexité élevée (il est exponentiel suivant le nombre de tâches de l'application) et ne permet pas de prendre en compte des critères qualitatifs supplémentaires.
   Il n'est donc pas utilisable dans la pratique, mais permet de mettre en évidence la calculabilité des arbres, et leurs caractéristiques.
   (d) le quatrième chapitre permet de confronter l'approche linéaire, par encapsulation, et notre approche.
   Nous montrons que : *si une application est ordonnançable avec les approches classiques linéaires, alors elle reste ordonnançable avec les approches arborescentes.*

Par contre, si elle n'est pas ordonnançable avec les approches classiques, on ne peut rien dire quant à son ordonnançabilité avec les approches arborescentes.
Nous illustrons également l'importance des contraintes structurelles que nous avons imposées ;

(e) le cinquième et dernier chapitre présente la méthode que nous utilisons dans la pratique pour obtenir les arbres d'ordonnancement.
Nous y présentons la modélisation complète d'un système de tâches en utilisant le formalisme des réseaux de Petri.
Les deux aspects du modèle de tâche sont considérés (arbre d'exécution et chemin) et comparés.
Une section de ce chapitre est consacrée à la mise en œuvre.
Nous y montrons comment, en partant du code des tâches temps-réel, nous produisons toutes les relations d'incompatibilité, puis, nous produisons le modèle de réseau de Petri.

En conclusion, dans le chapitre 9, nous précisons à nouveau nos objectifs initiaux, et synthétisons les résultats obtenus pendant la thèse.
Dans les perspectives immédiates, nous montrons comment le réseau de Petri construit va être utilisé pour générer les arbres d'ordonnancement et envisageons à long terme la génération des arbres d'ordonnancement répondant à des critères *qualitatifs*.

Première partie

État de l'art

Chapitre 2

Synthèse de quelques travaux connexes

"*Malgré soi, on est de son siècle*"
Auguste Comte

Sommaire

Nous présentons dans ce chapitre les approches que nous avons trouvées dans la littérature qui prennent en compte la sémantique (plus précisément les instructions conditionnelles) contenue dans le code des tâches temps-réel en vue de l'analyse d'ordonnançabilité des systèmes.
Pour chaque approche présentée, nous faisons une comparaison avec notre approche globale de résolution.
Cette comparaison portera sur la structure du code des tâches considérées, la modélisation de ces tâches et la technique d'analyse en vue de répondre au problème de la validation.
Nous verrons que dans la plupart des cas, les systèmes considérés vérifient des hypothèses plus restrictives ou ne coincident pas avec les nôtres.
Les auteurs n'intègrent pas les relations **intra-tâches et inter-tâches** à la phase de modélisation, et leur objectif est de fournir un diagnostic (une conclusion d'ordonnançabilité) de l'application pendant la phase d'analyse et de validation du système, pas de donner une description du fonctionnement effectif.
Nous avons décomposé ce chapitre en deux parties.
Dans la première partie, nous présentons quelques approches qui prennent en compte

les tests conditionnels de façon générale, mais sans que la prise en compte des instructions conditionnelles fasse l'objet d'une étude spécifique.
Nous ne détaillons pas ces approches, et en donnons juste une vue succincte.
Dans la deuxième partie, celle la plus proche de notre travail, nous présentons des travaux qui intègrent explicitement les instructions conditionnelles, et dont l'objectif en est l'étude pendant les phases de modélisation et d'analyse.

2.1 Prise en compte implicite des conditionnelles

De façon générale, le modèle de base de [Liu 1973] a aujourd'hui largement évolué, et les auteurs proposent de plus en plus des modèles de systèmes temps-réel beaucoup plus précis.
Par conséquent, on trouve dans la littérature plusieurs travaux qui proposent différentes approches d'intégration d'éléments visant à rendre le modèle plus proche de la réalité, dans le but d'optimiser l'ordonnancement.
Dans cette section, nous présentons brièvement quelques unes de ces approches.

2.1.1 Une approche basée AADL et réseaux de Petri

L'objectif des travaux de **Renault et Al.** [Renault 2009a] [Renault 2009b] est la validation des aspects temporels, structurels et comportementaux des applications temps-réel.
L'application à étudier est modélisée en **AADL** (Architecture Analysis and Design Language). Il s'agit d'un langage de conception et d'analyse des systèmes, dont les points clés sont présentés dans [SAE 2008].
C'est un langage à base de composants (matériels, logiciels et systèmes) qui permet de représenter aisément la concurrence, le parallélisme, les relations de précédence dans les applications temps-réel.
On trouve dans la littérature plusieurs outils qui permettent de valider les systèmes temps-réel modélisés suivant ce langage.
On peut citer [Out] : **Ocarina**, **Osate/Eclipse**, **Stood**. Ces outils de modélisation intègrent des techniques de validation classiques : pire temps de réponse, **RM**, **ED**.
L'une des particularités de l'approche de **Renault et Al.** est la validation des aspects comportementaux et structurels.
Les auteurs transforment un modèle **AADL** donné en un modèle de réseau de Petri coloré et temporisé. Ils utilisent ensuite toute la puissance des réseaux de Petri pour valider les aspects qui les intéressent.
La sémantique considérée dans leur approche est déduite des interactions entre les différents composants **AADL** du modèle initial. Il ne s'agit donc pas de relations entre les tests des tâches de l'application.

2.1.2 Une approche basée UML

Les travaux de **Phan et Al.** [Phan 2004a] [Phan 2004b] exploitent la connaissance interne des tâches d'une application modélisée suivant le formalisme **UML** [OMG 2001], afin d'éliminer les scénarios (cas d'utilisation suivant UML) impossibles, en accord avec un cahier de charges donné, et de proposer un ordonnancement plus précis qui tient compte des contraintes spécifiées dans le cahier de charges.
Ces travaux sont connus sous le nom de la méthode **Accord/UML**.
Cette méthode permet à un non expert du domaine temps-réel de construire des spécifications complètes du système qu'il veut modéliser, d'automatiser son développement et de conclure quant à son ordonnançabilité.
Les différentes étapes dans la validation d'une application suivant **Accord/UML** sont :

1. *Modèle d'Analyse Préliminaire* : présentation des principaux cas d'utilisation ;
2. *Modèle d'Analyse Détaillé* : présentation en détail du modèle structurel, du modèle d'interaction et du modèle de comportement ;
3. *Modèle de Prototypage* : construction d'un exécutable avec mise en évidence des contraintes temps-réel ;
4. *Modèle d'Analyse d'Ordonnançabilité* : modélisation des aspects temporels pour l'analyse d'ordonnançabilité.

L'approche globale consiste donc tout d'abord à étendre le formalisme **UML** (Unified Modeling Language) de base, afin d'y inclure de nouveaux profils.
Les applications sont ensuite modélisées suivant le formalisme étendu (modèle **UML** à objets temps-réel).
On peut générer leur pseudo-code et en faire une validation temporelle et comportementale.
La sémantique considérée ici provient du *Modèle d'Analyse Détaillé*, qui contient les diagrammes d'activité, et le *Modèle d'Analyse d'Ordonnançabilité* est transmis à l'outil **Agatha** [Aga] pour validation.
L'outil **Agatha** a été initialement développé pour vérifier la cohérence entre une spécification et un cahier de charges donné.
Il permet entre autres d'obtenir le graphe d'exécution d'une spécification donnée.
Il est donc possible d'obtenir l'ensemble des comportements de l'application modélisée.
Une commande *SCHEDULE* a été ajoutée au modèle afin d'implémenter un algorithme d'ordonnancement des tâches.
L'algorithme implémenté dans le cadre de leurs travaux est un algorithme à priorité dynamique, qui choisit à chaque instant la tâche dont l'échéance est la plus petite.
Il faut toutefois noter que n'importe quel autre algorithme aurait pu être implémenté.
Notons pour terminer ce paragraphe sur les approches basées **UML** que, afin d'unifier l'ensemble des approches basées **UML** (**SDL**, **Accord/UML**, **MAST** ...), l'*Object Management Group* a normalisé le profil **UML MARTE** [Mar] pour la

modélisation et l'analyse des systèmes temps-réel embarqués.

Les approches présentées section 2.1.1 et 2.1.2 s'intéressent à la validation comportementale des applications, soit en vérifiant que le système donné est valide d'un point de vue structurel, soit qu'il est bien en accord avec le cahier de charges.
Les tests conditionnels sont pris en compte dans ces approches d'une part à travers les instructions de choix dans le réseau de Petri, et d'autre part à travers les diagrammes d'activité dans l'approche *UML*.
Mais, à notre connaissance, l'objectif initial des auteurs n'est pas l'étude de la prise en compte de ces instructions conditionnelles durant la phase de validation.
C'est la raison pour laquelle il serait difficile de faire une comparaison de ces approches avec la notre.

Dans les approches qui suivent, le but du travail est la prise en compte **explicite** des instructions conditionnelles contenues dans le code des tâches des applications manipulées, durant les phases de modélisation et de validation.

2.2 Prise en compte explicite des conditionnelles

2.2.1 Le modèle quasi-statique

Les travaux de **Sgroi et Al.** [Sgroi 1999] utilisent une approche mixte (statique et dynamique) de validation structurelle des applications temps-réel : certaines décisions sont prises **hors-ligne** (pré-emption, suspension ...), et d'autres **en-ligne** (celles portant sur les instructions conditionnelles if ... then ... else, while ... do).
L'approche proposée, *quasi-static scheduling*, part d'un ensemble de spécifications contenant des instructions, dont des conditionnelles et des boucles, représenté par un réseau de Petri, et l'objectif est d'obtenir une décomposition en un nombre minimum de tâches, dont ils déterminent de plus les taux d'exécution relatifs (par exemple 1 fois la tâche 1 pour 2 fois la tâche 2 ...).
Leur problème est donc :

1. l'identification des tâches ;
2. l'obtention du code de chaque tâche.

Mais ils ne traitent pas des aspects plus temps-réel. En particulier, les instructions de contrôle temps-réel ne sont pas considérées.
Leur point commun avec notre approche est qu'ils construisent en fait, sous le nom de *schedule*, un objet très proche de nos arbres qui suit la règle : un préfixe, un choix (les branches étant construites sur le même mode).
Leur *schedule* constitue l'ensemble des différentes branches de l'arbre, et leur condition garantit qu'on peut reconstruire l'arbre, c'est à dire qu'il n'y a pas de branches manquantes.

2.2.2 Les conditional task graphs

Les *conditional task graphs* sont utilisés par [Shin 2003] [Lombardie 2010] pour modéliser aisément les contraintes de précédences complexes entre les tâches.
Elles permettent de prendre en compte de façon explicite les relations conditionnelles entre les tâches d'une application.
Leur objectif est de pouvoir répondre de façon plus fine à la question : le système est-il ordonnançable ou pas ?
Pour cela, après une étape de modélisation des précédences, ils proposent un algorithme d'ordonnançabilité qui intègre toutes les conditions.

2.2.2.1 Structure et modélisation

Dans cette approche, une tâche est caractérisée par les quatre paramètres de [Liu 1973] r, C, D et P, et une application temps-réel périodique est représentée par un **graphe de tâches conditionnelles** (**CTG**).
Ce graphe noté $G = < V, E >$ est un graphe orienté acyclique où V est l'ensemble des tâches et E est l'ensemble des arcs conditionnels orientés entre les tâches. Chaque arc indique que la tâche antérieure doit être exécutée entièrement, avant que la nouvelle tâche ne commence son exécution. À chaque arc $e = (T_i, T_j)$ est associée une condition c et une probabilité d'activation $Prob(c)$ associée au résultat de la condition.
Dans le cas où il n'y a pas de conditionnelles, la probabilité d'activation est 1.
Lorsqu'un nœud T_i possède plusieurs branches conditionnelles, c'est à dire plusieurs possibilités d'exécutions T_j $(i \neq j)$, à chaque branche T_j est associée une condition c_{ij}, et le résultat de la condition détermine la tâche qui s'exécutera.
Si la condition c_{ik} est vérifiée, la tâche T_k s'exécutera quand la tâche T_i aura terminée son exécution.
À notre connaissance, rien n'est dit dans la littérature quant au choix de la branche à suivre lorsque plusieurs conditions sont vérifiées.
Notons pour conclure cette partie sur la modélisation des *conditional task graphs* qu'une tâche peut avoir plusieurs nœuds prédécesseurs.
Dans ce cas, certains nœuds sont des **and-node** (la tâche successeur attend la terminaison de deux ou plusieurs tâches avant de s'exécuter), et d'autres nœuds sont des **or-node** (il suffit qu'une seule des tâches prédécesseurs soit terminée, pour que la tâche successeur s'exécute).

Nous donnons figure 2.1 un exemple de représentation d'une application suivant l'approche *conditional task graphs*.
Cette application comporte 7 tâches périodiques.
L'exécution des tâches T_2 et T_3 ne peut commencer qu'après l'exécution complète de la tâche T_1.
Une seule est exécutée, le choix dépendant du résultat du test c_1.
Les nœuds T_4 et T_6 sont en relation avec le nœud T_7 par un **or-node**, puisque les tâches T_4 et T_6 sont en exclusion mutuelle.

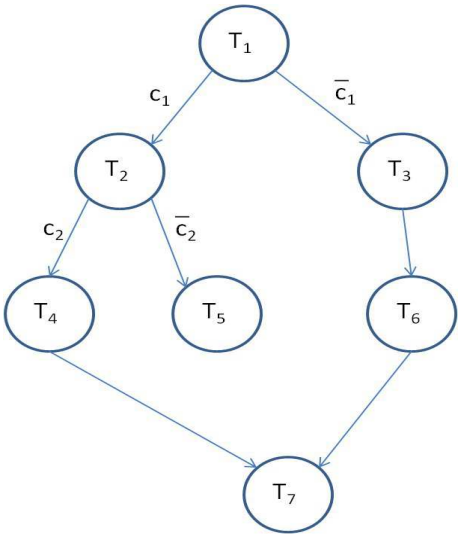


Figure 2.1 – Modélisation d'un système avec des relations de précédence suivant l'approche *conditional task graphs*

Par conséquent, il suffira que T_4 ou T_6 soit terminée, pour que la tâche T_7 soit activée.

2.2.2.2 Validation

Pour valider les applications ainsi modélisées, les auteurs proposent un algorithme d'ordonnançabilité à priorité dynamique, qui intègre la configuration du graphe représenté.
Nous ne présentons pas cette approche dans ce rapport, car seules celles permettant d'avoir une vue du comportement du système (les approches hors-ligne) nous motivent.
Le lecteur intéressé peut consulter la littérature [Shin 2003] [Lombardie 2010].

2.2.2.3 Comparaison avec notre approche

Une différence principale avec notre approche est qu'ici, la structure interne des tâches n'est pas explicitement prise en compte.
Par conséquent, les instructions conditionnelles, les relations **intra-tâches** et **inter-tâches** ne peuvent pas être gérées : les conditionnelles sont vues à l'échelle macroscopique, et les tâches sont non préemptibles.
On peut certes ramener notre approche à la leur, en subdivisant nos tâches en sous-tâches constituées de blocs indépendants. Mais par la suite, l'objectif recherché n'est pas le même.
En effet, pendant que nous cherchons à obtenir le comportement de l'application durant son exécution, [Shin 2003] et [Lombardie 2010] utilisent un algorithme d'or-

donnancement à priorité dynamique pour conclure à l'ordonnançabilité effective de l'application ou pas.

2.2.3 Une approche objet en environnement distribué

Les travaux présentés dans cette section concernent une approche hors-ligne d'ordonnancement distribué des applications dont les tâches ont des relations de précédence et comportent des instructions conditionnelles.
L'idée principale [El.Rewini 1995] [Verhoosel 1996] [Santoshkumar 1998] de l'approche est de proposer un ordonnancement final de l'application qui consiste en un regroupement des ordonnancements obtenus en parcourant les différentes branches d'exécution provenant des conditionnelles.
L'objectif étant de pouvoir dire avant implémentation que le système peut être ordonnancé ou pas.

2.2.3.1 Le modèle de tâche

Les tâches considérées sont périodiques et à échéance sur requête (le délai critique est donc égal à la période). Elles peuvent partager des ressources, et échanger des messages.
Le modèle utilisé est basé sur les approches probabilistes.
Chaque tâche est constituée d'un ensemble d'objets. Une tâche est donc représentée par un ensemble de branches (d'objets) en parallèle avec chacune une probabilité donnée d'être exécutée.
Chaque objet contient un ensemble de méthodes qui permettent de le manipuler, et chaque méthode est une combinaison de 1 ou plusieurs *bead*.
Un *bead* peut être soit un bloc d'exécution indépendant, soit une communication entre 2 méthodes.
Les *bead* contiennent la pire durée d'exécution de la tâche (celle relative au chemin sur lequel on se trouve).
Nous donnons figure 2.2 une vue de la représentation des tâches dans l'approche de **Santoshkumar et Al.**.

2.2.3.2 Validation

Les auteurs proposent une approche incrémentale de construction de la séquence d'ordonnancement.
Les séquences choisies tiennent compte de la probabilité du choix d'une branche d'exécution, mais aussi du délai critique de la tâche.
L'idée générale consiste à générer l'ensemble des cas d'exécution possibles.
Il s'agit de la combinaison de tous les chemins d'exécution possibles entre les tâches de l'application. Le nombre de cas possible est égal au produit de tous les chemins des tâches.
Pour chaque chemin d'exécution généré, les auteurs construisent une séquence d'ordonnancement en utilisant un algorithme d'ordonnancement hors-ligne déterministe.

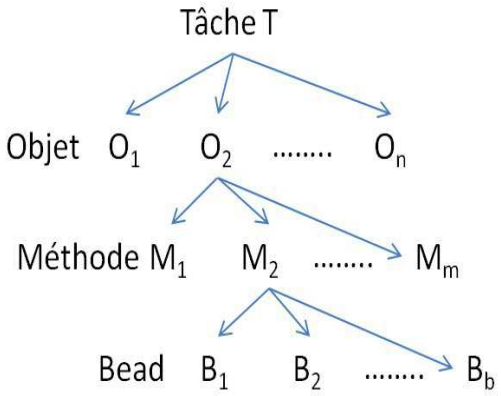


Figure 2.2 – Représentation d'une tâche T dans l'approche de **Santoshkumar et Al.**

Enfin, l'ensemble final des ordonnancements est obtenu en considérant toutes les séquences construites dans les différents chemins d'exécution.
Le principe est entièrement décrit dans la littérature. Nous ne le détaillons pas, car nous n'utilisons pas le modèle probabiliste.

2.2.3.3 Comparaison avec notre approche

Nous pouvons noter plusieurs similitudes entre ce travail et le notre :

1. tout d'abord la modélisation proposée va permettre, tout comme dans notre approche de considérer le corps des tâches ;
2. ensuite, pendant la validation, la prise de décision quant au choix d'exécution d'une branche est faite *en-ligne* ;
3. enfin, tout comme nous le verrons pour nos arbres d'ordonnancement, l'ordonnancement résultant contient tous les chemins possibles de la tâche pendant son exécution, bien que la notion d'arbre d'ordonnancement ne soit pas explicitement invoquée.

En fait, l'ensemble des ordonnancements obtenus dans ces travaux constituent ce que [Pailler 2006] a qualifié par la notion *d'ordonnançabilité locale* dans ses travaux (voir section 3.4 pour les travaux de **Pailler et Al.** et section 3.4.3 pour la notion d'ordonnançabilité locale), et il a montré que cette approche ne suffit pas pour une étude d'ordonnançabilité.
Par contre, notre travail ne porte pas sur les systèmes distribuées, et nous n'avons pas trouvé dans l'état de l'art une application de ce travail aux systèmes monoprocesseur, comme c'est le cas dans notre étude.
Enfin, notre approche n'est pas probabiliste.
C'est la raison pour laquelle, pendant la validation, nous produisons un arbre d'exécution qui contient tous les chemins possibles, alors que dans leur approche, le choix

des séquences dépend fortement de la probabilité d'exécution d'une branche conditionnelle.

2.2.4 Une approche probabiliste

L'approche de **Cucu-Grosjean** [Cucu 2009] concerne l'ordonnancement, en utilisant des approches probabilistes [Burns 2003], des tâches périodiques à priorité fixe en environnement monoprocesseur et/ou multiprocesseur pour lesquelles les temps d'exécution sont incertains.
Cette approche permet de considérer des modèles de tâches plus réalistes, en ce sens que les pires durées d'exécution sont moins surévaluées.

2.2.4.1 Le modèle de tâche

Chaque tâche T_i est caractérisée par les paramètres classiques de [Liu 1973] : sa date de premier réveil r_i, son délai critique D_i et sa période P_i.
La particularité dans son approche concerne le paramètre C_i.
Il est donné comme un ensemble de durées d'exécution c_k, chacune avec sa probabilité $P(C = c_k)$ d'être obtenue.

$$C_i = \begin{pmatrix} c_k \\ P(C = c_k) \end{pmatrix}_{k \in [1,\ldots,k_i],\ k_i \in \mathbb{N}^*}$$

Le paramètre c_k est compris entre un c_i^{min} et un c_i^{max} qui sont respectivement les valeurs minimales et maximales de la durée d'exécution de la tâche T_i.
Dès que le modèle est ainsi formalisé, à chaque réveil de la tâche, une seule durée d'exécution va être choisie.

L'exemple 2.2.4.1 illustre une modélisation de **Cucu-Grosjean** pour un système de trois tâches, T_1, T_2 et T_3 où T_1 est déterministe et T_2 et T_3 probabilistes.

Exemple 2.2.4.1 : Un système de 3 tâches modélisé suivant **Cucu-Grosjean**
$T_1 = (0, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, 3, 3), T_2 = (3, \begin{pmatrix} 2 & 3 & 4 \\ 0.5 & 0.1 & 0.4 \end{pmatrix}, 6, 6)$ et $T_3 = (2, \begin{pmatrix} 1 & 2 \\ 0.5 & 0.5 \end{pmatrix}, 6, 6)$.
$C_1 = 1$, quelque soit l'instance considérée. C'est le cas classique du modèle de [Liu 1973].
Une instance de la tâche T_2 peut avoir pour durée d'exécution 2, avec une probabilité de 0.5, une durée d'exécution égale à 3, avec une probabilité de 0.1, ou une durée d'exécution égale à 4 avec une probabilité égale à 0.4.
Le même raisonnement s'applique sur la tâche T_3.

2.2.4.2 La technique de validation

Pour valider un système ainsi modélisé, l'auteur doit vérifier que toutes les tâches respectent leurs échéances, indépendamment de la durée d'exécution choisie.
Seuls les ordonnancements préemptifs à priorités fixes sont envisagés.

La méthodologie de validation est basée sur une extension du calcul du temps de réponse des tâches [Diaz 2002] [Cucu 2006].
Le calcul du pire temps de réponse est affiné afin d'intégrer les durées d'exécution variables et leur probabilité.
L'auteur calcule les probabilités d'obtention du pire temps de réponse par rapport aux branches choisies. Il a ainsi une conclusion d'ordonnançabilité probabiliste, qui dépend des variables en entrée des temps d'exécution.
Les auteurs montrent que leur méthode d'obtention du pire temps de réponse est moins complexe que les approches classiques qu'on trouve dans [Diaz 2002] [Cucu 2006].

2.2.4.3 Comparaison avec notre approche

Pour faire une analogie avec notre approche, nous pouvons supposer que les différentes durées d'exécution d'une tâche correspondent, dans notre cas, aux différents choix des comportements conditionnels de la tâche, avec la probabilité correspondante.
Par contre, les tâches considérées ont des priorités fixes, et rien n'est dit sur l'échange de messages ou le partage des ressources.
De même, la structure des tâches n'étant pas explicitement prise en compte, rien n'est dit sur les relations (au sens **inter-tâches**) pouvant exister entre les tâches de l'application.
Les durées d'exécution des tâches sont données par des variables aléatoires indépendantes. Les relations **inter-tâches** ne sont donc pas gérées.
La classe des systèmes considérés est donc incluse, tout comme la classe des systèmes considérés dans les approches précédentes, dans celle que nous considérons.
Enfin, le fait de donner une conclusion d'ordonnançabilité pour valider le système, bien qu'elle soit plus réaliste que les approches du pire temps de réponse classiquement utilisées, fait perdre à la fin la notion de comportement qu'on espérait faire ressortir à travers l'ensemble de multi-durées d'exécution.
Notre approche le mettra explicitement en évidence.

2.2.5 Le modèle de tâches récurrents

Dans un contexte où les tâches sont indépendantes, le *modèle de tâches récurrent* de **Baruah** [Baruah 2003] prend en compte de façon explicite les instructions conditionnelles.
L'approche de **Baruah** peut être vue comme une extension des modèles multiframes [Aloysius 1996] [Baruah 1999] et sporadiques [Liu 1973] [Mok 1983], et utilise des algorithmes à priorités statiques ou dynamiques pour conclure quant à l'ordonnançabilité des systèmes étudiés.

2.2.5.1 La structure des tâches

La structure générale des tâches est décrite figure 2.3.

```
For(;;) - - instruction caractérisant la périodicité de la tâche
{
        When(évènement extérieur déclencheur);
        La tâche s'exécute pendant une durée e0 avec une deadline d0;
        if(C) /*la condition C dépend de l'état du système, de l'environnement, de
                la date d'activation de l'évènement déclencheur, et ne peut pas
                être modifiée par la tâche elle-même */
        then
                La tâche s'exécute pendant une durée e1 avec une deadline d1;
        else
                La tâche s'exécute pendant une durée e2 avec une deadline d2;
}
```

Figure 2.3 – Structure des tâches considérées par **Baruah**

Une tâche est constituée d'instructions conditionnelles ($if \ldots then \ldots else \ldots endif$) et d'instructions indépendantes (e_i).
Les branches des instructions conditionnelles peuvent contenir des instructions indépendantes ou des instructions conditionnelles.
Chaque instruction indépendante s'exécute pendant une durée e_i, et doit se terminer avant une durée d_i après son activation.
Nous pouvons remarquer que comme nous l'avons souligné dans nos hypothèses (section 1.2.2.1), la condition C dans ce cas dépend de l'environnement, et pas de variables qui pourraient être modifiées par la tâche elle-même.

2.2.5.2 La modélisation

Le *modèle récurrent* permet de représenter de façon explicite tous les comportements d'une tâche donnée sous la forme de la figure 2.3.
Dans le *modèle récurrent* de **Baruah**, une tâche T est représentée par un graphe $G(T)$ et une période $P(T)$.
Le graphe $G(T)$ est un graphe acyclique orienté avec un unique sommet de départ et une unique feuille.
Chaque sommet du graphe représente une *sous-tâche* (une sous-tâche au sens de **Baruah** est un bloc d'exécution d'une tâche), et chaque arc un flux de contrôle possible.
Chaque sommet u est étiqueté par deux paramètres entiers $e(u)$ et $d(u)$: à chaque instant où la sous-tâche u est exécutée, une instance de tâche est générée avec pour date de réveil la date de début d'exécution de la sous-tâche, pour durée d'exécution $e(u)$ et pour délai critique $d(u)$.
Si u et v sont deux sommets de la tâche modélisée, l'arc (u, v) est étiqueté par un

entier $p(u, v)$ qui correspond à la durée minimale entre le moment où l'origine du lien est activée et l'activation du nœud pointé par le lien.
Nous donnons figure 2.4 une illustration de l'approche de modélisation de **Baruah**.

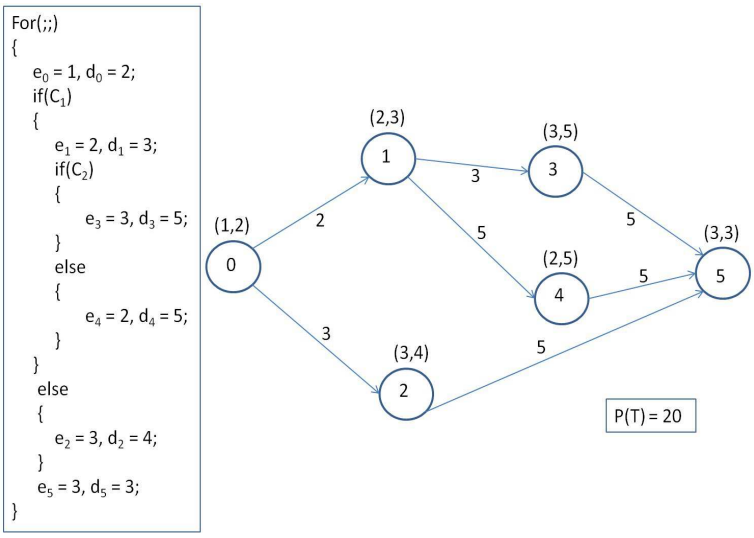


Figure 2.4 – Un exemple de modélisation d'une tâche par le modèle récurrent

La tâche T est périodique de période 20, et possède des instructions conditionnelles.
Le nœud 0 s'exécute en 1 unité de temps, et doit être entièrement exécuté après 2 unités de temps.
Après l'exécution du nœud 0, le nœud 1 ou 2 peut être activé, cela avec un minimum de 2 ou 3 unités de temps . . ..
L'exécution du nœud 5 met fin à l'exécution de la tâche, et le cycle recommence à la prochaine ré-activation de T.

2.2.5.3 L'approche de validation

L'idée générale est d'améliorer les techniques d'analyse classiques basées sur la *Demand Bound Function* et la *Request Bound Function* [Buttazo 1997] [Baruah 2003], en intégrant le *modèle récurrent*.
Baruah propose ainsi une condition nécessaire et suffisante d'ordonnançabilité à priorités dynamiques des applications modélisées suivant son modèle récurrent (voir [Baruah 2003]).
Cette condition est plus précise que dans le cas général.
En effet, elle prend en compte toutes les branches qui peuvent être suivies par la tâche.
Le test d'ordonnançabilité est par conséquent plus satisfaisant (parce qu'il est plus précis) et moins complexe : l'approche présentée par **Baruah** a dans le cas général la même complexité pseudo-polynomiale que les approches classiques sporadiques,

multiframes.
Toutefois, pour les applications temps-réel pour lesquelles certaines tâches sont modélisées par un graphe orienté acyclique, l'approche de **Baruah** est plus adaptée.

2.2.5.4 Comparaison avec notre approche

Tout d'abord au niveau de la structure, les tâches ne sont pas décrites de la même façon.
Dans l'approche de **Baruah** une tâche est subdivisée en plusieurs sous-tâches, avec chacune une durée d'exécution et un délai critique, alors que les tâches que nous manipulons ont un seul délai critique global.
Pour utiliser ce modèle, il nous faudrait ajouter des contraintes d'échéances intermédiaires à notre modèle, ce qui en modifierait le comportement.
De plus, rien n'est dit sur la prise en compte explicite des dates de premier réveil.
Dans le cas où les tâches sont à départs différés, le *modèle récurrent* ne montre pas comment elles seront prises en compte.
Ce problème est par contre explicitement présenté dans notre approche.
Cette approche portant sur les tâches indépendantes, la classe des systèmes étudiés est moins grande que la notre, puisque notre travail porte sur les tâches qui peuvent échanger des messages, partager des ressources ....
Le graphe orienté acyclique utilisé pour la modélisation des tâches permet de les représenter dans le détail, et d'avoir ainsi une vue du comportement effectif des tâches.
Par contre, nous n'avons pas l'impression qu'il est ensuite utilisé pendant l'analyse de l'ordonnançabilité.
La condition nécessaire et suffisante d'ordonnançabilité utilise en effet des paramètres qui s'obtiennent tous à partir du code des tâches.
Notre approche est basée sur une génération des ordonnancements à partir du modèle du système.
Enfin, la technique de validation utilisée (basée sur **EDF** [Serlin 1972] [Liu 1973]) n'est pas optimale dans notre contexte [Dertouzos 1974] [Labetoulle 1974].
De plus, **Baruah** donne une condition nécessaire et suffisante d'ordonnançabilité améliorée, sans produire de modèle d'exécution. Elle n'est pas adaptée dans notre cas, où, rappelons le, nous sommes intéressés par le comportement de l'application durant son exécution.
Notons enfin, pour conclure avec les approches utilisant le *modèle récurrent*, qu'il existe aussi les travaux de **Madhukar et Al.** [Madhukar 2008], plus récents, qui peuvent être vus comme une extension des travaux de **Baruah**.
Les modèles qui y sont proposés (*recurring branching task model* et *recurring task model with branching and control variables*) offrent plus de possibilités que celui de **Baruah** en prenant en compte plus de paramètres.
Ils permettent de prendre explicitement en compte le contrôle des variables, les ressources ....
Nous ne les présentons pas, puisqu'ils s'inscrivent dans la même logique.

2.2.6 L'approche OASIS

L'approche présentée dans cette section a été mise en œuvre dans le projet **OASIS**.
Le but de ce projet [Aussaguès 1998] [Touzalin 2001] [Aussaguès 2002] est d'offrir une plateforme permettant la conception multitâches temps-réel déterministe d'une application de sûreté.
L'idée générale consiste en la proposition d'une méthodologie de conception des systèmes critiques tout en garantissant le respect des contraintes temporelles de l'application et en préservant le déterminisme de son comportement global.

2.2.6.1 Structure des tâches

Les taches temps-réel du projet *OASIS* ont un code qui s'exécute indéfiniment, encadré par les instructions FOR_EVER et END_FOR.
Ce sont donc des tâches qui peuvent être périodiques ou pas, constituées d'une séquence finie d'instructions de types :

1. instructions conditionnelles classiques :
   $IF(c) \ldots THEN \ldots (ELSE \ldots) END\_IF$, où c est la condition ;
2. instructions $ADV(n)$ pour la gestion du temps, où n est une constante entière ;
3. instructions $SEND/RECEIVE$ pour la gestion explicite des communications. Précisément l'envoi/réception des messages.
   Il existe aussi un mécanisme implicite de gestion de communications à travers des variables temporelles ;
4. et enfin on trouve des portions de codes indépendantes sans instructions de type 1, 2 et 3.

Une illustration de la structure d'une tâche suivant **Aussaguès et Al.** est donnée figure 2.5.

```
T1
FOR_EVER
Trait1
IF (Cond)
THEN
    Trait2
    ADV(k1)
ELSE
    Trait3
    ADV(k2)
ENDIF
END_FOR
```

Figure 2.5 – Structure d'une tâche du projet **OASIS**

La tâche T_1 est périodique[1], et possède une instruction conditionnelle.
Elle exécute d'abord un bout de code indépendant ($Trait_1$), ensuite une instruction conditionnelle.
Si la condition $Cond$ est satisfaite, la tâche exécute un bout de code $Trait_2$, cela avant qu'il ne se soit écoulé $k_1 * H_1$ unités de temps depuis le dernier point de contrôle (H_1 caractérise l'horloge locale de la tâche T_1), et exécute de nouveau l'instruction $Trait_1$.
Si par contre la condition $Cond$ n'est pas respectée, la tâche exécute le bout de code $Trait_3$, jusqu'à ce que $k_2 * H_1$ unités de temps se soient écoulées depuis la dernière activation, puis la tâche est ré-activée ....

2.2.6.2 Modélisation

Le modèle d'une tâche *OASIS* est composé de deux parties :

1. une partie pour la gestion des aspects temporels.
   Une **horloge locale** qui est associée à chaque tâche et définit les instants physiques de traitement des entrées et de production des sorties, et l'**horloge globale** associée au système qui est l'union de toutes les horloges locales de toutes les tâches de l'application.
   La manipulation de chaque horloge temps-réel se fait par la primitive **ADV(k)** où k correspond au nombre d'instants de l'horloge qui doivent s'écouler avant la reprise des traitements de la tâche considérée ;
2. une partie pour la gestion des communications et du corps de la tâche.
   La communication se fait par l'envoi de messages asynchrones (boîtes aux lettres) et/ou à travers des variables temporelles.
   Lorsqu'elle se fait par envoi de messages, à chaque message est associée une date de visibilité à partir de laquelle le message est dans la boîte aux lettres de la tâche consommatrice.

Deux modèles de tâches peuvent être utilisés, suivant que l'objectif est la vérification de la ponctualité (il s'agit du respect des contraintes temporelles), ou bien la vérification des propriétés de sûreté :

1. s'il s'agit de vérifier la ponctualité, le modèle utilisé est la représentation des tâches en graphe d'états-transitions.
   Chaque tâche T_i est représentée par un graphe d'états-transitions $G_i = (X_i, U_i)$, où l'ensemble des nœuds du graphe X_i décrit les états observables de la tâche et l'ensemble des arcs U_i décrit les transitions entre deux états observables.
   Ainsi, un nœud représente une instruction ADV, l'arc entrant (sortant) représente la portion de code qui la précède (qui la suit).
   Nous donnons figure 2.6 une illustration de cette modélisation.
   Nous pouvons remarquer (sur la figure 2.6) qu'à l'activation, la tâche commence par exécuter $Trait_1$, IF, $Trait_2$.

[1] La notion de périodicité ici est sensiblement différente de la notre, puisque dans ce cas, la période n'est pas unique, mais dépend du choix du comportement de la tâche

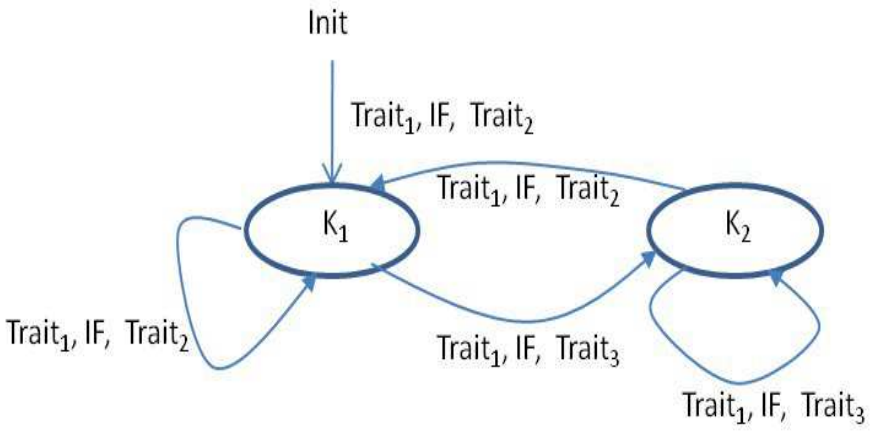


Figure 2.6 – Modélisation en graphe d'états-transitions de la tâche de la figure 2.5

Ensuite elle peut suivre soit le chemin $Trait_1$, IF, $Trait_2$ avant que k_1 unités de temps ne s'écoulent (par rapport à l'horloge de la tâche), soit le chemin $Trait_1$, IF, $Trait_3$ avant que k_2 unités de temps ne s'écoulent (par rapport à l'horloge de la tâche), puis, à partir de là, elle peut faire $Trait_1$, IF, $Trait_3$ avant que k_2 unités de temps ne s'écoulent (par rapport à l'horloge de la tâche), ou $Trait_1$, IF, $Trait_2$ avant que k_1 unités de temps ne s'écoulent (par rapport à l'horloge de la tâche) et ainsi de suite ;

2. s'il s'agit de vérifier les propriétés de sûreté de fonctionnement de l'application, qui tiennent compte de la dynamique du système, **Aussaguès et Al.** utilisent une modélisation basée sur les réseaux de Petri autonomes :
   (a) le temps est modélisé par un réseau de Petri ordinaire où chaque transition représente les instants d'activation et où le nombre de marques dans les places représente le quota de temps ;
   (b) les autres traitements sont modélisés par des ensembles de places et de transitions où les transitions représentent l'exécution de ces traitements et où les marques dans les places représentent la position dans le code.

   Nous ne détaillons pas cette approche.
   Notons juste qu'elle permet elle aussi de prendre explicitement en compte les instructions conditionnelles et les aspects temporels de toutes les tâches de l'application.

2.2.6.3 Validation

Deux aspects de la validation sont considérés :

1. s'il s'agit de vérifier la ponctualité du système, les auteurs utilisent le modèle de tâches états-transitions.
   Ils construisent le produit synchronisé des tâches constituant l'application à

valider.
Le graphe de ce produit est ensuite valué en utilisant les valeurs des graphes de chaque tâche.
Le lecteur intéressé par la construction et la valuation du graphe du produit synchronisé peut consulter la littérature.
Notons ici pour conclure que le parcours complet du graphe du produit synchronisé des tâches donne un ensemble d'inéquations et d'équations à satisfaire pour que le système soit valide.
D'autre part, il contient toutes les interactions possibles entre tous les traitements de toutes les tâches, et représente donc le comportement du système pendant son exécution ;

2. si la validation concerne les propriétés de sûreté, les auteurs utilisent le modèle de tâche basé sur les réseaux de Petri.
Ils peuvent ainsi à travers la recherche d'invariants ou l'analyse du graphe des marquages mettre en évidence des erreurs comportementales.

2.2.6.4 Comparaison avec notre approche

De façon générale, **Aussaguès et Al.** proposent une approche de validation hors-ligne des applications temps-réel basée modèle, qui intègre explicitement les instructions conditionnelles et les aspects temporels.
Bien que sur le principe cela soit la même approche que nous considérons, certaines différences peuvent être soulignées :

1. nous étudions dans la structure des tâches la prise en compte des ressources.
De plus, dans notre modèle les messages sont supposés toujours reçus, et la réception est bloquante ;
2. le départ différé des tâches est explicitement pris en compte.
De plus, l'approche de validation dans **OASIS** est uniquement basée sur **ED** [Serlin 1972] [Liu 1973], ce qui n'est pas notre cas ;
3. dans notre modèle, les aspects sémantiques inter-tâches sont explicitement pris en compte ;
4. enfin, la résolution du système d'équations et d'inéquations déduites du graphe du produit synchronisé donne une condition d'ordonnançabilité de l'application, mais ne donne pas à l'aide d'un arbre comme dans notre cas, le comportement effectif de la tâche pendant son exécution.
Il est *probablement* possible d'en extraire les arbres, mais les auteurs n'ont pas, à notre connaissance, étudié ce cas.

Bilan

La liste des travaux intégrant la sémantique afin de proposer un ordonnancement optimal des tâches temps-réel est peut être plus longue.
Nous en avons cité quelques uns dans ce chapitre.

De façon plus spécifique, dans le cadre de cette thèse, *la sémantique concerne les aspects liés aux instructions conditionnelles dans le code des tâches.*
L'état de l'art que nous avons trouvé a été discuté dans ce chapitre.
Il en ressort que :

1. les classes des systèmes dans les autres approches sont en général incluses dans notre classe de système considérée.
   Cela veut dire que les tâches que nous considérons ont plus de propriétés et sont plus générales que les tâches considérées dans la plupart des systèmes ;
2. les modèles proposés n'intègrent pas le plus souvent la sémantique au sens **intra-tâche**, et nous n'en avons trouvé aucun qui l'intégrait au sens **inter-tâches** ;
3. dans la plupart des cas, les techniques de validation sont en-ligne, et consistent en une condition d'ordonnançabilité, qui, bien qu'elle soit moins complexe, ne permet pas d'avoir le comportement du système pendant son exécution.
   Et dans les cas où les auteurs proposent une validation hors-ligne, ils utilisent la notion de séquence d'ordonnancement, qui elle aussi ne permet pas d'avoir le comportement effectif du système quand les tâches possèdent des instructions conditionnelles.
   Nous n'avons pas trouvé dans la littérature une approche qui permet d'avoir les arbres d'ordonnancement que nous proposons dans ce rapport.
   La structure de **Sgroi et Al.** en est proche, mais elle n'intègre pas le temps, et les ordonnancements de **Santoshkumar et Al.** peuvent être étendus pour les ressembler, particulièrement en ajoutant les relations **inter-tâches** ;
4. enfin, dans le même ordre d'idées, nous n'avons pas trouvé d'approche qui considère de façon explicite les relations entre les tâches, au sens **inter-tâches** que nous avons définis.

L'intérêt de notre travail est double :

1. tout d'abord nous proposons un modèle de tâches qui intègre une classe de tâches très générales : départs simultanés ou différés, envoi et/ou réception de messages, prise et/ou libération de ressources, prise en compte des relations sémantiques ... ;
2. ensuite nous approfondissons l'étude de faisabilité du système.
   Nous ne nous contentons plus de dire que le système est valide, mais lorsque c'est le cas, nous proposons tous les ordonnancements qui ont un sens.
   De plus, chacun des ordonnancements, c'est à dire chaque arbre d'ordonnancement, permettra d'avoir une vue réelle du comportement de l'application pendant son exécution.

Nous présentons dans le chapitre 3 une approche de validation de systèmes temps-réel, qui sera l'approche que nous utiliserons, qui permet de couvrir ces deux aspects.

Chapitre 3

Validation par les réseaux de Petri

"Il est plus aisé, et éminemment plus scientifique, de traquer le passé que d'esquisser l'avenir"
Dion Jean

Sommaire

Dans le chapitre 2, nous avons répertorié les approches existantes qui prennent en compte les instructions conditionnelles, en vue de la validation des applications temps-réel.
Comme nous l'avons souligné en conclusion du chapitre 2, ces approches traitent en partie certains de nos problèmes, mais restent en général assez éloignées de nos objectifs.
Une option consisterait à proposer des artéfacts et méthodes pour compléter ces approches. C'est un travail dont nous n'avons pas évalué l'ampleur, mais nous pensons que la tâche serait pénible parce que, d'une part les modèles ont été choisis pour des cas bien précis, et d'autre part nous ne disposons pas du temps nécessaire.
Dans ce chapitre, nous présentons les approches existantes dans notre laboratoire. Ces approches ont été développées dans le même contexte d'étude que notre travail. Par la suite, nous les complèterons pour pouvoir résoudre notre problématique.

Nous présentons donc dans ce chapitre une technique de validation hors-ligne des systèmes temps-réel basée sur les réseaux de Petri, puis nous montrons comment cette technique a été complétée pour intégrer les tâches avec des instructions conditionnelles.
Nous concluons en faisant ressortir à travers notre problématique, le travail qu'il reste à faire, que nous avons résolu dans cette thèse.
Nous commençons par présenter le modèle de réseaux de Petri que nous utilisons.

3.1 Les réseaux de Petri

Les réseaux de Petri ont initialement été proposés en 1962 par [Petri 1962] pour l'étude dynamique des systèmes complexes.
Il s'agit d'un outil graphique et mathématique pour spécifier, modéliser, comprendre et valider les systèmes complexes.
La littérature est assez fournie pour le lecteur intéressé par l'étude et la manipulation des réseaux de Petri [Peterson 1981] [Best 1987] [Choquet-Geniet 1993] [Jensen 1997] [Choquet-Geniet 2006].
Dans cette section, nous présentons uniquement les termes et aspects dont nous aurons besoin.
Nous utilisons dans cette thèse les notations et définitions des réseaux de Petri proposées dans [Best 1987] [Grolleau 1999] [Choquet-Geniet 2000] [Grolleau 2002] [Choquet-Geniet 2006] [Pailler 2006].

Définition 3.1.1 Réseau de Petri
Un **réseau de Petri** (encore appelé **réseau place-transition**) **marqué** est un couple (N, M_0) où $N = (Q, T, W)$ $(Q \cap T = \emptyset)$ avec :

1. Q : ensemble fini de places ;
2. T : ensemble fini de transitions ;
3. $W : QXT \cup TXQ \Rightarrow \mathbb{N}$ la fonction de valuation.
   W permet de valuer l'arc entre une place et une transition, ou entre une transition et une place ;
4. $M_0 : Q \Rightarrow \mathbb{N}$ le marquage initial du réseau.
   $M_0(p)$ est le nombre de marques (jetons) contenues(s) dans la place p.

N fournit une description statique d'un système, par exemple une configuration de tâches pouvant interagir, M_0 décrit son état initial.
La dynamique du système est décrite par la règle de tir des transitions.

Définition 3.1.2 Validité d'une transition
Une transition $t \in T$ est **valide** pour un marquage M si et seulement si $\forall\ p \in Q$, $M(p) \geq W(p, t)$.
Le **tir de cette transition** conduit au marquage M' défini par :
$\forall\ p \in Q,\ M'(p) = M(p) - W(p, t) + W(t, p)$.
Nous notons $M(t > M'$.

La définition 3.1.2 s'étend de manière naturelle aux séquences de transitions. Cette dynamique du réseau nous conduit à définir la notion de marquages accessibles.

Définition 3.1.3 Marquages accessibles
L'ensemble de tous les **marquages accessibles** à partir d'un marquage M_0 dans un réseau N, noté $Acc(N, M_0)$ est :
$Acc(N, M_0) = \{M \ tel \ que \ \exists \ t_{i1}, \ldots, t_{ik} \in T, \ , k \in \mathbb{N}, \ M_0(t_{i1}, \ldots, t_{ik} > M\}$.
Il s'agit intuitivement de l'ensemble de tous les marquages qu'on peut atteindre à partir d'un marquage M_0 donné.

Définition 3.1.4 Langage d'un réseau
Le **langage d'un réseau marqué** est l'ensemble des séquences valides depuis le marquage initial.

Afin de pouvoir étendre la notion de transition valide à un ensemble de transitions, nous commençons par introduire la notion de conflit effectif, qui intuitivement signifie que des transitions ne peuvent pas être tirées au même moment.

Définition 3.1.5 Conflit effectif
Notons ${}^o t$ l'ensemble des places p telles que $W(p, t)$ est non nul.
Deux transitions t_1 et t_2 sont en **conflit effectif** pour un marquage M si
$\exists \ p \in {}^o t_1 \cap {}^o t_2$ tel que $W(p, t_1) + W(p, t_2) > M(p)$.

Nous définissons ensuite la notion de fonctionnement d'un réseau sous la règle du tir maximal qui permet d'étendre la notion de tir à un ensemble de transitions.

Définition 3.1.6 Règle de tir maximal
Un réseau de Petri fonctionne sous **la règle de tir maximal** si et seulement si à chaque tir, on franchit un ensemble maximal, au sens de l'inclusion, de transitions simultanément valides et telles que aucune paire de transitions de l'ensemble ne soit en conflit effectif.

Notons que la puissance d'expression des réseaux fonctionnant sous la règle de tir maximal est la même que celle des réseaux temporisés [Ramchandani 1974] fonctionnant au plus tôt [Starke 1990].

Un exemple de fonctionnement d'un réseau fonctionnant sous la règle du tir maximal, avec mise en évidence des conflits est donné figure 3.1.
Les transitions en conflit effectif dans ce réseau sont t_1 et t_2 ou t_2 et t_3.
Ce réseau fonctionnant sous la règle du tir maximal, les ensembles maximaux possibles sont $\{t_2\}$ et $\{t_1, \ t_3\}$.
Les nouveaux réseaux obtenus après un tir sont donnés figure 3.1 selon l'ensemble maximal tiré.

Par la suite, nous utiliserons les réseaux de Petri colorés [Jensen 1997].
Notons que dans ce rapport, la définition des réseaux de Petri colorés que nous utilisons est extrêmement simplifiée.

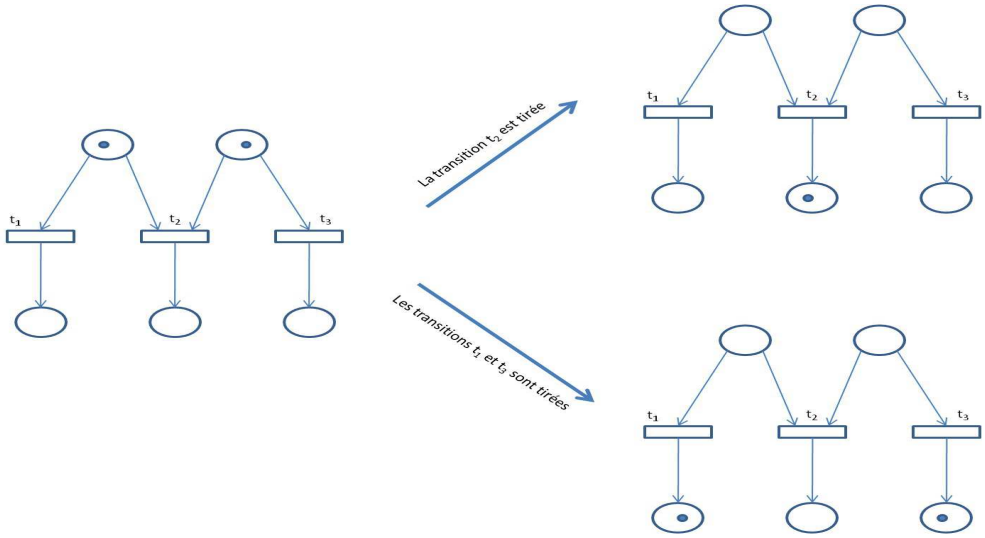


Figure 3.1 – Fonctionnement d'un réseau sous la règle du tir maximal avec gestion de conflits

En effet, dans la définition générale, les couleurs sont associées aux transitions et aux places, alors que dans notre thèse, nous ne les associerons qu'aux places.
C'est la raison pour laquelle nous parlerons de réseaux de Petri P-colorés.

Définition 3.1.7 Réseau de Petri P-coloré
Un **réseau de Petri P-coloré marqué** est un triplet (N, M_0, C) où $N = (Q, T, W)$ $(Q \cap T = \emptyset)$ avec :

1. C : un ensemble fini de couleurs ;
2. $W : QXTXC \cup TXQXC \Rightarrow \mathbb{N}$ la fonction de valuation ;
3. $M_0 : QXC \Rightarrow \mathbb{N}$ le marquage initial ;
4. la **règle de tir** est définie par : une transition $t \in T$ est valide à partir du marquage M si et seulement si $\forall p \in Q$ et $\forall c \in C$, $M(p,c) \geq W(p,t,c)$.
   Le tir de cette transition conduit au marquage M' défini par : $\forall\ p \in Q, \forall\ c \in C$, $M'(p,c) = M(p,c) - W(p,t,c) + W(t,p,c)$.

Afin de contrôler l'évolution du réseau, on peut le contraindre par un **ensemble terminal** [Cartensen 1984] [Valk 1981].
Il s'agit d'un ensemble de marquages du réseau, vérifiant une ou plusieurs propriété(s) définie(s) sur les marquages.
Cela nous amène à définir la notion de centre du langage terminal.

Définition 3.1.8 Centre du langage terminal
L'ensemble des séquences pour lesquelles les marquages successifs atteints restent dans l'ensemble terminal est appelé le **centre du langage terminal**.

Afin d'avoir une vue simple de l'évolution du réseau, on peut construire son graphe de marquages.

Définition 3.1.9 Graphe de marquages
Le **graphe de marquages** d'un réseau de Petri est un graphe étiqueté orienté dont les nœuds sont les marquages accessibles, et un arc d'étiquette t relie un marquage M à un marquage M' si et seulement si $M(t > M'$.

Remarquons que la construction effective du graphe de marquages d'un réseau de Petri nécessite qu'il soit borné.

Définition 3.1.10 Réseau de Petri borné
Un **réseau de Petri est borné** si, pour tout marquage accessible du marquage initial, le nombre de marques (jetons) contenues dans chaque place du réseau est borné (en fait est inférieur à un entier k : on parle dans ce cas de **réseau k-borné**). De façon équivalente, cela signifie que le nombre de marquages accessibles est borné.

Un réseau de Petri peut être **étiqueté**.
Dans ce cas, le graphe de marquages obtenu est par conséquent lui aussi étiqueté.

Définition 3.1.11 Réseau de Petri étiqueté
Un **réseau de Petri étiqueté** est un triplet $< N, \Sigma, \sigma >$ où :

1. N est le réseau ;
2. Σ est un alphabet ;
3. $\sigma : T \Rightarrow \Sigma$ est un morphisme d'étiquetage.

Plusieurs types d'arcs peuvent être utilisés pour relier les places aux transitions. Dans cette thèse, nous utiliserons entre autres les arc inhibiteurs.

Définition 3.1.12 Arc inhibiteur
Un **arc inhibiteur** (voir notation figure 3.2) est un arc entre une place et une transition, avec la condition que la transition est valide si et seulement si la place d'entrée est vide.

Nous donnons figure 3.2 un exemple de réseau de Petri avec un arc inhibiteur.

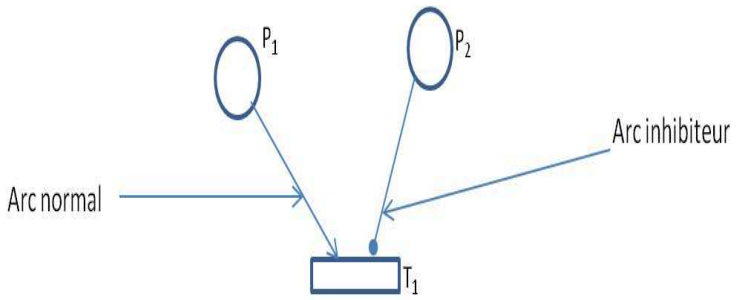


Figure 3.2 – Réseau de Petri avec arc inhibiteur : T_1 ne peut être franchie que si P_1 a au moins 1 jeton et P_2 est vide

Enfin, nous aurons besoin de réseaux de Petri synchronisés.
Un réseau de Petri est dit synchronisé lorsque des évènements extérieurs sont associés à une action.
Il peut par exemple s'agir d'actions ayant lieu à une date précise qui se trouve alors associées au tir de la transition ainsi étiquetée.

Définition 3.1.13 Réseau de Petri synchronisé
Un **réseau de Petri synchronisé** est un quadruplet $(N, M_0, E, Sync)$ où
$N = (Q, T, W)$ $(Q \cap T = \emptyset)$ avec :

1. (N, M_0) le réseau de Petri marqué ;
2. E un ensemble d'évènements externes ;
3. $Sync : T \Rightarrow E \cup \{e\}$ où e est l'évènement toujours occurrent, c'est à dire qu'il se produit en permanence.

Dans un réseau de Petri synchronisé, une transition est donc valide si elle est valide pour le marquage, mais elle n'est tirée effectivement que lorsque l'évènement auquel elle est associée se produit.

Nous illustrons figure 3.3 la notion de réseau de Petri synchronisé.

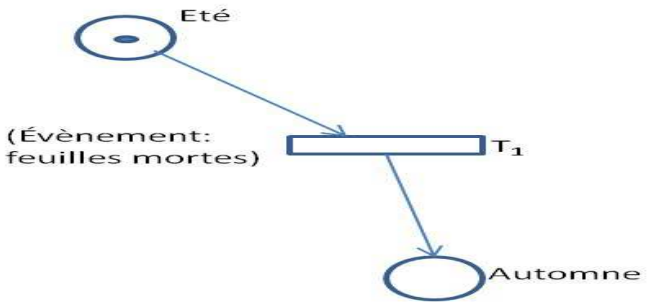


Figure 3.3 – Réseau de Petri synchronisé : le passage de l'été à l'automne dépend non seulement du fait qu'on soit en été (un jeton dans la place Été), mais est synchronisé sur l'évènement apparition de feuilles mortes

Nous choisissons dans ce rapport, d'utiliser une approche de validation hors-ligne basée sur les réseaux de Petri, pour résoudre notre problématique.
Ce choix est justifié section 3.2.

3.2 Justification des choix

Pour résoudre les problèmes posés dans cette thèse (voir section 1.2), deux choix ont été faits :

1. celui d'utiliser les approches hors-ligne pour l'analyse des systèmes ;
2. celui d'utiliser les réseaux de Petri pour la modélisation et l'étude des systèmes.

3.2.1 Du choix des approches hors-ligne

Dès que les tâches d'un système temps-réel partagent des ressources et/ou sont liées par des contraintes de précédence, les approches en-ligne ne sont plus les plus performantes.
En effet, dans le cas général, il n'existe pas de stratégie en-ligne optimale [Dertouzos 1989], c'est à dire une stratégie d'ordonnancement qui ordonnance correctement toute application ordonnançable, car le problème de l'ordonnancement est NP-dur [Baruah 1990].
Les approches hors-ligne ont une puissance d'ordonnancement supérieure aux approches en-ligne.
En effet, elles s'appuient sur la construction exhaustive de toutes les séquences valides. Donc, s'il existe au moins une telle séquence, elle sera détectée.
De plus, ces approches permettent d'obtenir une description précise du comportement de l'application.
Le prix à payer pour cette puissance d'ordonnancement est une complexité élevée.
Cependant, la complexité de l'énumération des séquences valides peut être améliorée grâce à de bonnes heuristiques pendant la recherche de ces séquences.
Il existe plusieurs techniques d'analyse basées sur les approches hors-ligne [Baker 1974] [Bratley 1975] [Lawler 1981] [Martel 1982] [Xu 1992] [Tindell 1992] [Xu 1993] [Beauvais 1996] [Cavalcante 1997] [Zamorano 1997] [Beauvais 1998] [Largeteau 2002] [Grolleau 2002] [Largeteau 2005] [Laalaoui 2005].
Nous choisissons d'utiliser dans cette thèse une technique basée sur le modèle de réseau de Petri.
Nous justifions ce choix dans la section 3.2.2.

3.2.2 Du choix des réseaux de Petri

De la définition faite section 3.1 sur les réseaux de Petri, nous pouvons remarquer qu'ils offrent entre autres la possibilité de modéliser aisément l'échange de messages, le partage des ressources, le parallélisme, la concaténation et le choix d'instructions, les blocs pré-emptifs ou non.
De plus, en définissant un *bon* alphabet sur un réseau de Petri, le graphe des marquages obtenu correspond à l'ensemble des comportements valides de l'application modélisée. On peut donc extraire de ce graphe l'ensemble des ordonnancements valides du système.
Les réseaux de Petri, en particulier ceux temporisés, offrent la possibilité de prendre en compte le temps.
De même, les réseaux de Petri que nous utiliserons, avec la règle du tir maximal, permettent de gérer les contraintes temporelles [Grolleau 1999] [Choquet-Geniet 2000] [Grolleau 2002] [Pailler 2006] [Choquet-Geniet 2006].
L'avantage d'utiliser notre modèle (réseau de Petri sous la règle du tir maximal) par rapport à un modèle temporisé est sa simplicité de représentation et surtout d'implémentation.

De plus, il correspond à un fonctionnement où la durée d'une action élémentaire est constante, ce qui correspond à notre hypothèse.

Pour conclure cette section sur la justification des choix, l'approche choisie nous permettra de modéliser et d'analyser les systèmes temps-réel de façon détaillée, en intégrant les contraintes temporelles et comportementales de l'application.
Cette approche a déjà été utilisée par **Grolleau et Al.** et **Pailler et Al.** dont nous présentons les travaux dans les sections 3.3 et 3.4.

3.3 Modélisation des tâches sans instructions conditionnelles

Les travaux de **Grolleau et Al.** [Grolleau 1999] [Choquet-Geniet 2000] [Grolleau 2002] présentent une méthodologie d'étude hors-ligne d'applications temps-réel constituées de tâches dépendantes.
Afin d'illustrer cette approche, nous présentons tout d'abord le système de tâches sur lequel sont basés ces travaux, ensuite l'approche de modélisation utilisée pour les systèmes manipulés et enfin la technique de validation considérée.

3.3.1 Le système de tâches

L'application est représentée comme un ensemble fini de tâches $\{T_1, \ldots, T_n\}$ périodiques issues d'une étude du cahier des charges [Stankovic 1998].
Le squelette d'une tâche est constitué de :

1. blocs fonctionnels écrits dans un langage de haut niveau ;
2. primitives temps-réel de prise/libération de ressources ;
3. primitives temps-réel d'émission/réception de messages.

Chaque tâche T_i est temporellement caractérisée par les 4 paramètres temporels r_i, C_i, D_i et P_i définis section 1.1.3.1.

Un exemple de pseudo code de tâche manipulée est donné figure 3.4.

3.3.2 Modélisation

Le modèle utilisé par **Grolleau et Al.** consiste en un réseau place/transition avec ensemble terminal et fonctionnant sous la règle du tir maximal.
Il comporte deux parties (voir figure 3.5) :

1. un réseau modélisant le **système de tâches**, où chaque transition est associée à une action.
   Le système étant par hypothèse monoprocesseur, une seule tâche s'exécute à la fois.
   Il y a donc une place *processeur*, contenant un unique jeton, en entrée/sortie de chacune des transitions, qui assure l'exclusion mutuelle entre les différentes

Tâche *UneTâche*
r = *sa date de réveil*;
C = *sa pire durée d'exécution*;
D = *son délai critique*;
P = *sa période*;
Début
b_1 = *durée d'exécution d'un bloc fonctionnel*;
Send (Message); *instruction d'envoi d'un message. La durée de l'envoi est contenue dans le bloc* b_1
b_2 = *durée d'exécution d'un bloc fonctionnel;*
Lock(Ressource); *instruction de prise d'une ressource*
b_3 = *durée de l'utilisation de la ressource;*
Unlock(Ressource); *instruction de libération d'une ressource. Les durées de prise/libération de la ressource sont contenues dans* b_3
Receive(Message); *instruction de réception d'un message. La durée de réception du message est contenue dans* b_3
Fin.

Figure 3.4 – Exemple de tâche dans l'approche de **Grolleau et Al.**

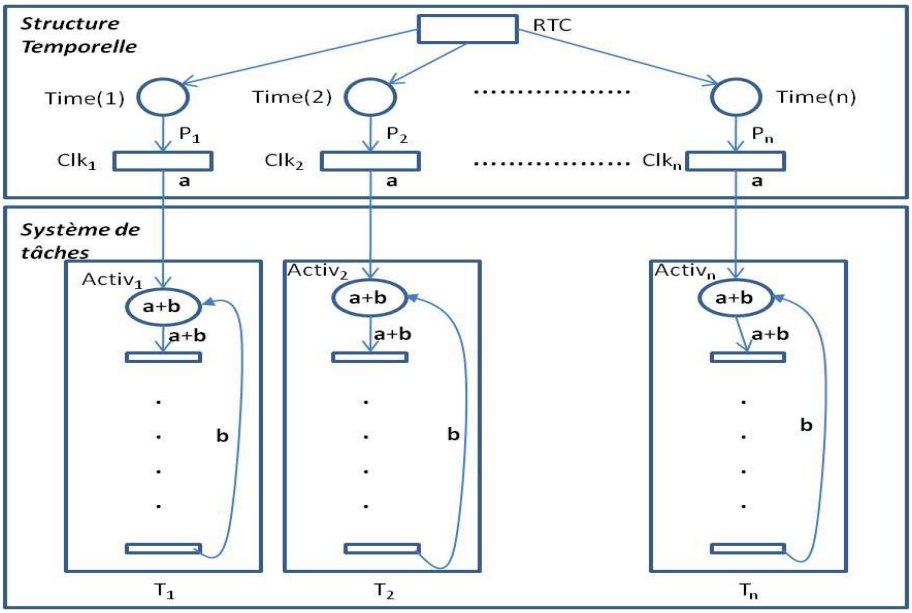


Figure 3.5 – Modélisation dans l'approche de **Grolleau et Al.** d'une application temps-réel

actions effectuées par les tâches.
Nous ne l'avons pas représentée pour une meilleure lisibilité de la figure 3.5. Par la suite, nous ne la représenterons pas.
Ensuite chaque tâche est représentée de manière classique par un ensemble de transitions mises en séquence.
Il y a une transition par unité de temps d'exécution des tâches.
De plus, afin de limiter la taille du modèle, les blocs indépendants (fonctionnels) sont modélisés suivant la technique présentée figure 3.6.
Concrètement, les blocs fonctionnels dont la durée d'exécution est supérieure ou égale à 3 sont condensés en un ensemble de 3 transitions, et ceux donc la durée est inférieure ou égale à 2 sont explicitement représentés.

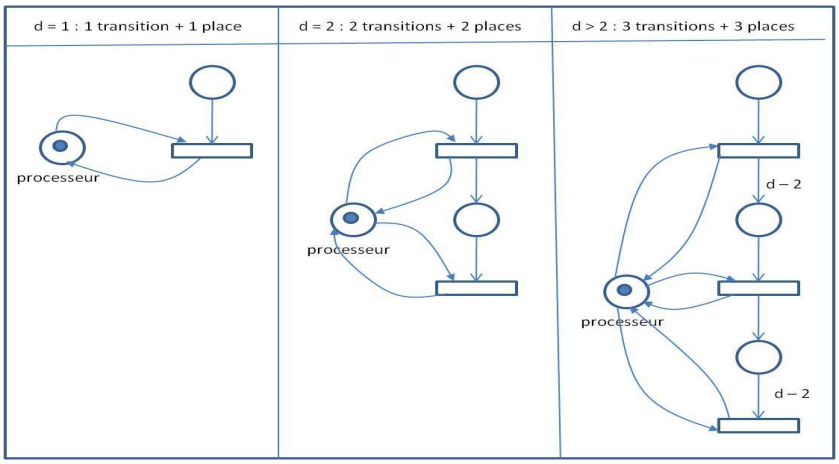


Figure 3.6 – Modélisation d'un bloc fonctionnel de durée d avec représentation de la place processeur

Les communications entre les tâches sont représentées à l'aide de places de type **boîte à lettres**.
Chaque boîte à lettre est modélisée par une place.
L'émission d'un message par une tâche vers cette boîte est modélisée par le dépôt d'un jeton dans cette place, et la réception par la consommation d'un jeton.
La modélisation d'une ressource se fait grâce à une place contenant initialement autant de jetons que la ressource comporte d'instances.
La prise de la ressource se traduit par la consommation d'un jeton, et sa libération par la remise d'un jeton dans cette place.
La première place de chaque tâche T_i nommée $Activ_i$, est une place colorée associée à un ensemble de deux couleurs notées **a** et **b**.
Un jeton de couleur **a** est déposé à chaque activation de la tâche, et un jeton de couleur **b** indique qu'aucune instance de la tâche n'est en cours d'exécution.
Lorsque la dernière action de la tâche s'exécute, la transition associée tire et dépose un jeton de couleur **b** dans la place $Activ_i$.

Une tâche ne peut commencer son exécution que si la place $Activ_i$ contient un jeton de couleur **a** et un jeton de couleur **b**.
Ce que nous représentons par **a**+**b**.
Notons qu'ainsi, la condition de la non ré-entrance des tâches est également modélisée ;

2. un réseau modélisant la **structure temporelle**.
   Ce réseau comporte :
   (a) une **horloge globale** notée RTC (Real-Time Clock), modélisée par une transition source.
   Comme le réseau fonctionne sous la règle de tir maximal, cette transition sera tirée à chaque tir de transitions.
   Elle va ainsi temporiser le fonctionnement du réseau, en produisant à chaque tir un jeton dans les horloges locales aux tâches.
   Cela crée une représentation logique du temps, une unité de temps d'exécution étant représentée par un tir de la transition RTC ;
   (b) une **horloge locale** pour chaque tâche T_i.
   Chaque horloge est composée d'une place $Time(i)$ permettant de comptabiliser le temps écoulé depuis la dernière activation de la tâche (jetons déposés par RTC), et d'une transition Clk_i, qui tire toutes les P_i unités de temps, produisant alors un jeton de couleur **a** dans la place $Activ_i$.

Pour prendre en compte les **dates de premier réveil** de chaque tâche T_i, **Grolleau et Al.** utilisent les marquages initiaux des places $Activ_i$ et $Time(i)$.
Ces marquages sont définis par :

$$M_0(Time(i)) = \begin{cases} 1 \ si \ r_i = 0 \\ P_i - r_i + 1 \ si \ r_i > 0 \end{cases}$$

et

$$M_0(Activ_i) = \begin{cases} \mathbf{a}+\mathbf{b} \ si \ r_i = 0 \\ \mathbf{b} \ si \ r_i > 0 \end{cases}$$

avec $r_i \leq P_i$.
Ces marquages caractérisent les places $Time(i)$ et $Activ_i$ selon que la date de réveil de la tâche T_i soit différée ou nulle.
Pour simplifier les descriptions, nous n'avons pas considéré le cas $r_i > P_i$, qui est traité dans [Grolleau 2002]. Nous pouvons cependant, si besoin est, intégrer cette modélisation à notre étude.
Afin de considérer les **délais critiques**, les auteurs définissent un ensemble terminal $\mathcal{I}$ qui garantit que toutes les tâches respectent leurs échéances.
Cet ensemble $\mathcal{I}$ est défini par :

$$M \in \mathcal{I} \Leftrightarrow \begin{cases} 1. \ M(Time(i)) > D_i \Rightarrow M(Activ_i) = \mathbf{b}; \\ 2. \ M(Time(i)) = 1 \Rightarrow M(Activ_i) = \mathbf{a}+\mathbf{b} \ ou \ M(Activ_i) = \mathbf{b}. \end{cases}$$

où les points 1 et 2 formalisent les contraintes qui assurent le respect des délais critiques :

1. si la place $Time(i)$ contient plus de D_i jetons, deux cas peuvent se produire :
   (a) soit la tâche T_i n'a pas encore été activée pour la première fois, et nous sommes donc dans un cas de tâches à départs différés, par conséquent la place $Activ_i$ contient toujours le jeton **b** déposé par le marquage initial, donc $M(Activ_i) = \mathbf{b}$;
   (b) soit la tâche T_i a déjà été activée au moins une fois, ce qui signifie que depuis la dernière activation, D_i unités de temps se sont écoulées. L'instance en cours doit donc avoir terminé son exécution, ce qui se traduit par la présence du jeton **b** dans la place $Activ_i$;
2. si la place $Time(i)$ contient 1 jeton, nous pouvons étudier deux cas :
   (a) soit la tâche T_i n'a pas encore été activée pour la première fois, nous sommes dans le cas où $M(Activ_i) = \mathbf{b}$;
   (b) soit elle a déjà été activée et elle a respecté son échéance, dans ce cas, elle vient d'être réactivée, et la place $Activ_i$ contient les jetons **a** et **b**.

3.3.3 Validation

Dans une approche hors-ligne, la validation d'une application consiste à fournir une séquence d'ordonnancement valide.
Pour obtenir toutes les séquences valides, les auteurs construisent le graphe des marquages terminaux.
Ce graphe est ensuite valué par l'alphabet défini par la fonction :

$$\psi : \begin{cases} \textit{une transition du système de tâches} \Rightarrow \textit{la tâche dont elle est issue} \\ RTC \Rightarrow \varepsilon \\ Clk_i \Rightarrow \varepsilon,\ \forall\ i \in \{1, \ldots, n\} \end{cases}$$

où ε est le mot vide.
Le centre du langage terminal du réseau de Petri est l'ensemble de toutes les séquences d'ordonnancement valides.
Les auteurs appliquent ensuite des techniques d'extraction et d'optimisation afin de trouver toutes les séquences valides.
Ces techniques ont été abondamment décrites dans **Grolleau et Al.**.
Nous ne les présentons pas, cela ne constituant pas l'objet de ce travail de thèse.

3.3.4 Synthèse de l'approche

L'approche complète de **Grolleau et Al.** pour l'obtention des séquences d'ordonnancement à partir du code de l'application est illustrée figure 3.7.
Le point d'entrée est un système de tâches donné sous sa représentation en pseudo-code, pour lesquelles les paramètres temporels (r, C, D, P) sont spécifiés.
La méthodologie d'étude consiste en trois étapes :

1. modélisation du système par réseaux de Petri avec marquages terminaux et fonctionnant sous la règle du tir maximal ;

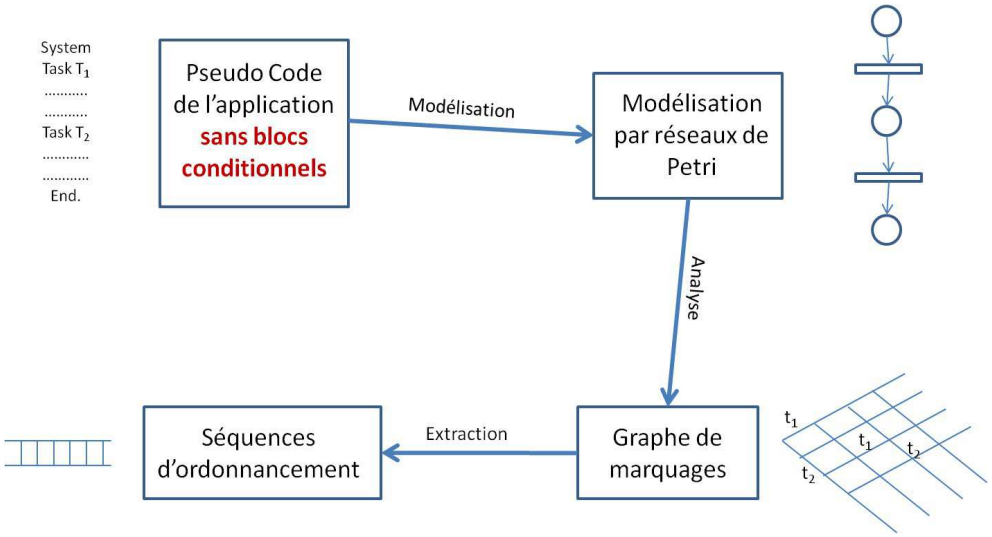


Figure 3.7 – Illustration de la méthodologie d'analyse des applications dans l'approche **Grolleau et Al.**

2. construction du graphe des marquages associé au réseau de Petri ;
3. construction de l'ensemble des séquences d'ordonnancement valides à partir du graphe des marquages, avec en plus possibilité d'extraction de certaines séquences optimales, suivant certains critères : par exemple les séquences dont le temps de réponse moyen est minimal.
   Ces critères seront par la suite qualifiés de critères qualitatifs.

3.3.5 Extensions de l'approche

L'approche de **Grolleau et Al.** permet de considérer une large classe de systèmes temps-réel pour une analyse d'ordonnançabilité : tâches à départ simultanés ou différés, tâches dépendantes à travers l'envoi et la réception des messages, le partage des ressources, les contraintes de précédences ....
Mais les tâches considérées soit ne comportent pas d'instructions conditionnelles, soit celles-ci ont été encapsulées.
Les travaux de **Pailler et Al.** ont eu pour objectif de permettre de les considérer de manière explicite.

3.4 Modélisation des tâches avec instructions conditionnelles

Les travaux réalisés par **Pailler et Al.** [Pailler 2004] [Pailler 2006] ont pour objectif de généraliser l'approche de **Grolleau et Al.** en intégrant les instructions conditionnelles dans les phases de modélisation et de validation.

En effet, lorsque cela n'est pas fait, nous rappelons que le comportement décrit est celui de la plus longue durée d'exécution, ce qui ne correspond pas nécessairement au comportement effectif de l'application.
Le modèle utilisé ne correspondant pas au fonctionnement de l'application, on ne peut donc pas en déduire de simulation du comportement effectif.
L'objectif des travaux de **Pailler et Al.** a en premier lieu consisté à reformuler le problème de l'ordonnancement dans le cas où les tâches peuvent comporter des instructions conditionnelles.
Comme en section 3.3 pour les travaux de **Grolleau et Al.**, nous commençons par présenter le nouveau système de tâches, qui prend explicitement en compte les instructions conditionnelles, puis nous montrons comment le modèle initial (celui de **Grolleau et Al.**) est modifié pour les considérer, et enfin nous présentons l'impact de cette extension sur la validation du système.

3.4.1 Le système de tâches

Comme précédemment, le corps des tâches est donné dans un langage de haut niveau (par exemple **C** ou **ADA**).
Le squelette de la tâche utilisé dans l'approche de **Grolleau et Al.** est complété afin de considérer les instructions conditionnelles.
Les tâches considérées dans cette approche ont donc en plus des **instructions conditionnelles**.
Concernant la modélisation temporelle, les trois paramètres temporels d'une tâche T_i (r_i, D_i et P_i) sont toujours considérés.
Par contre, pour pouvoir mettre en évidence les variations comportementales issues des instructions conditionnelles, le paramètre C_i est affiné.
Concrètement, il est étendu et remplacé par ζ_i qui correspond au multi-ensemble des durées des comportements de la tâche.
Chaque élément de l'ensemble ζ_i est en fait la pire durée d'exécution de l'un des comportements de la tâche T_i.
Le modèle temporel d'une tâche T_i est ainsi donné dans l'approche de **Pailler et Al.** par 4 paramètres : r_i, ζ_i, D_i et P_i.
Afin de conserver ce paramètre ζ_i pendant l'étude du système, l'approche utilisée dans **Pailler et Al.** consiste à représenter une tâche grâce à son arbre d'exécution comme illustré figure 3.8.
Nous pouvons remarquer que dans l'*approche arborescente* où on prend explicitement en compte l'instruction conditionnelle ($Test$), nous représentons tous les comportements (dans le cas de la figure 3.8 il y en a 2), suivant le résultat du test de la tâche.
Remarquons aussi que dans l'*approche classique*, le paramètre C correspond à la plus longue branche de la tâche.

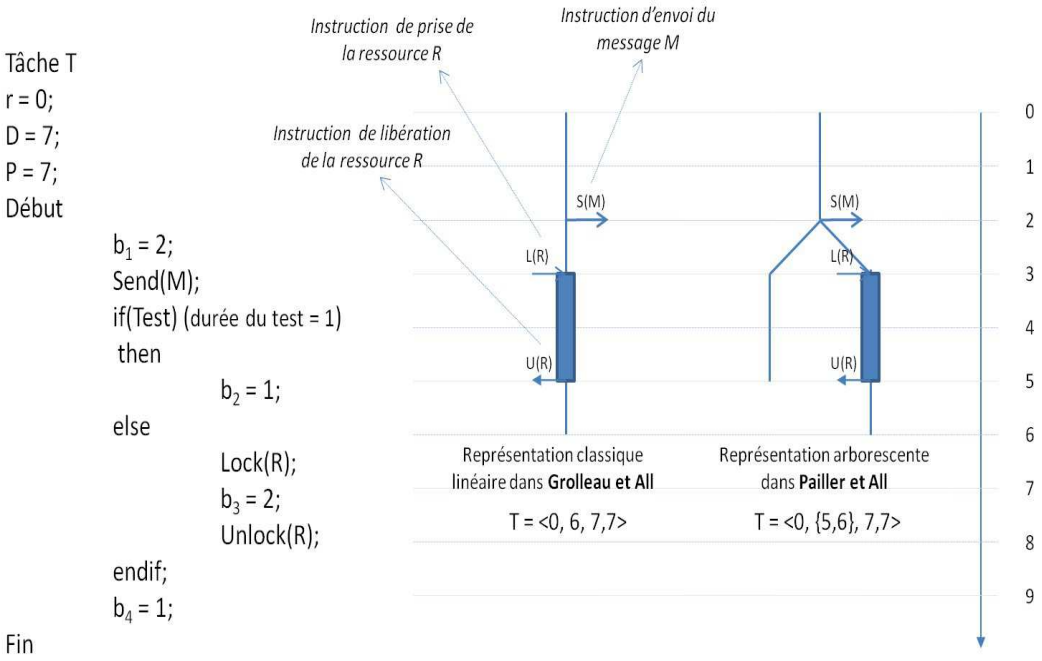


Figure 3.8 – Une tâche T avec une instruction conditionnelle, sa représentation classique, et sa représentation arborescente

3.4.2 Modélisation

L'application est modélisée comme un réseau de Petri avec ensemble terminal, fonctionnant sous la règle du tir maximal.
Comme dans le cas de **Grolleau et Al.**, la modélisation proposée comporte deux parties :

1. un réseau modélisant la **structure temporelle** qui est identique à celui présenté figure 3.5 et qui permet de représenter le temps ;
2. un réseau modélisant le **système de tâches**.
   Chaque tâche est toujours modélisée comme une séquence finie de places et de transitions.
   La principale innovation concerne la modélisation des blocs conditionnels.
   Elle se fait de façon classique (modélisation d'un choix avec les réseaux de Petri) par deux transitions concurrentes correspondant chacune à une alternative du test conditionnel.
   Le modèle de réseau de Petri d'une tâche conditionnelle va ainsi comporter autant de branches conditionnelles que de comportements de la tâche.
   La modélisation des primitives temps-réel, sous réserve qu'elles respectent les hypothèses énoncées section 1.2.2, est identique à l'approche utilisée dans **Grolleau et Al.**.
   L'avantage dans l'approche de **Pailler et Al.** est que, quand la primitive porte

sur une branche d'une conditionnelle, le modèle garantit bien qu'elle s'exécute sur cette branche. Ce qui n'est pas le cas dans l'approche de **Grolleau et Al.**.

Nous donnons figure 3.9(a) un exemple de modélisation d'une tâche avec prise en compte effective des instructions conditionnelles.

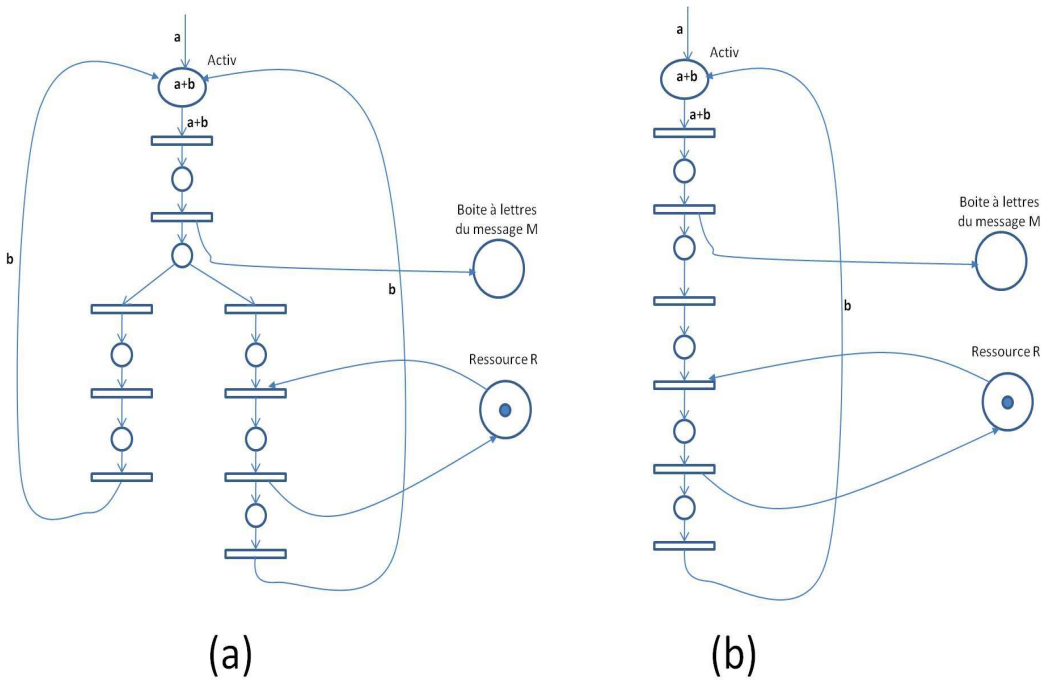


Figure 3.9 – Modélisation de la tâche T : approches de **Pailler et Al. (a)** et **Grolleau et Al. (b)**

La tâche modélisée est la tâche T de la figure 3.8.
Nous pouvons remarquer dans l'approche de **Grolleau et Al.** (figure 3.9(b)) l'utilisation de la ressource R pendant toutes les périodes de T, quand bien même ce ne serait pas le cas, du fait du choix de la branche conditionnelle alternative.

Notons enfin pour conclure cette partie sur la modélisation que les **ensembles terminaux** restent préservés (il y a bien à la fin de chaque tâche dépot du jeton **b** avant réactivation de la tâche).
Par conséquent, ce modèle gère de la même façon le respect des **délais critiques** que celui de **Grolleau et Al.**.
Il en est de même de la prise en compte des dates de **premier réveil**.

3.4.3 Validation

Pour considérer explicitement les branches conditionnelles durant la phase de validation, **Pailler et Al.** introduit la notion d'*arbre d'ordonnancement*.

Il s'agit d'un arbre dans lequel chaque branche correspond à un ordonnancement de l'application en ne considérant pour chaque instance de tâche qu'un seul chemin d'exécution.
Deux approches de validation d'applications en découlent :

1. une application va être dite **localement ordonnançable** si tous les chemins d'exécution de chaque tâche sont indépendamment ordonnançables.
   C'est à dire si toutes les séquences obtenues en considérant les différentes combinaisons des branches des tâches sont valides ;
2. une application va être dite **globalement ordonnançable** s'il existe au moins un arbre d'ordonnancement valide.

Pailler et Al. ont montré que dans le cas général, si une application est globalement ordonnançable, alors elle est localement ordonnançable.
L'inverse n'étant vrai que dans le contexte des tâches indépendantes.
Notre contexte étant celui des tâches dépendantes, nous nous sommes naturellement intéressés à la notion d'ordonnançabilité globale.
La validation dans ce cas repose sur l'étude du graphe des marquages du modèle proposé, pour la construction non plus des séquences d'ordonnancement, mais des *arbres d'ordonnancement* qui permettent de voir et comprendre tous les comportements du système.
Les arbres d'ordonnancement prennent en compte de façon explicite les branches conditionnelles qui apparaissent dans les arbres d'exécution des tâches.
Ils permettent ainsi de représenter pour chaque tâche tous ses comportements possibles, et d'avoir le comportement global de l'application.

Nous illustrons la notion d'arbre d'ordonnancement à travers l'exemple 3.4.3.1.

Exemple 3.4.3.1 Considérons une application S constituées de 3 tâches
$T_1 =< 0, \{3, 5\}, 15, 15 >$, $T_2 =< 0, \{4, 3\}, 15, 15 >$ et $T_3 =< 0, 2, 5, 5 >$ dont le code est donné figure 3.10.
Une représentation graphique de ces tâches est donnée figure 3.11 et nous donnons figure 3.12 un arbre d'ordonnancement de l'application S.
Remarquons qu'il contient tous les comportements qui peuvent être choisis par l'application S.
Ces différents comportements proviennent des différents choix effectués dans les conditionnelles.
Comme nous l'avons dit, ces arbres d'ordonnancement seront extraits du graphe de marquages.
Les techniques d'extraction et d'optimisation sont précisées dans les travaux de **Pailler et Al.**. Elles étendent celles utilisées dans les travaux de **Grolleau et Al.**.
Pour conclure, notons que l'approche utilisée dans les travaux de **Pailler et Al.** est bien une extension des travaux de **Grolleau et Al.**.
Par conséquent, dans le cas où la(les) tâche(s) n'a(ont) pas d'instruction conditionnelle, la modélisation et la validation du système est la même que dans le cas de **Grolleau et Al.**.

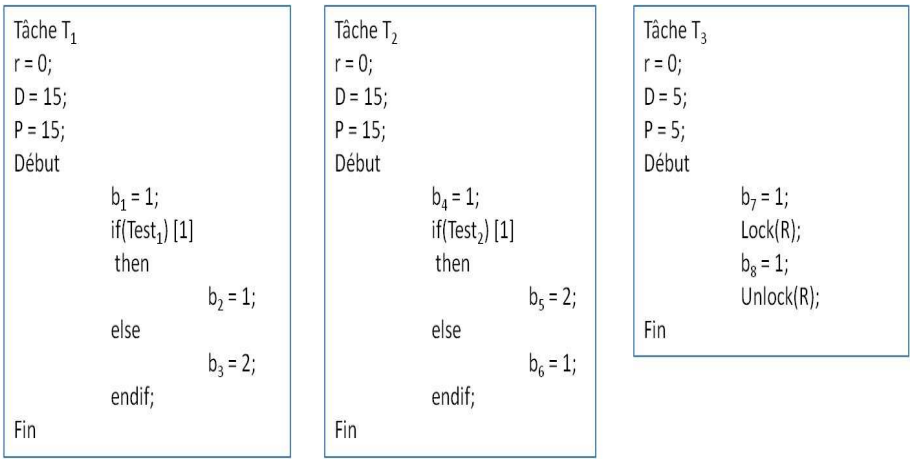


Figure 3.10 – Tâches utilisées pour illustrer l'arbre d'ordonnancement

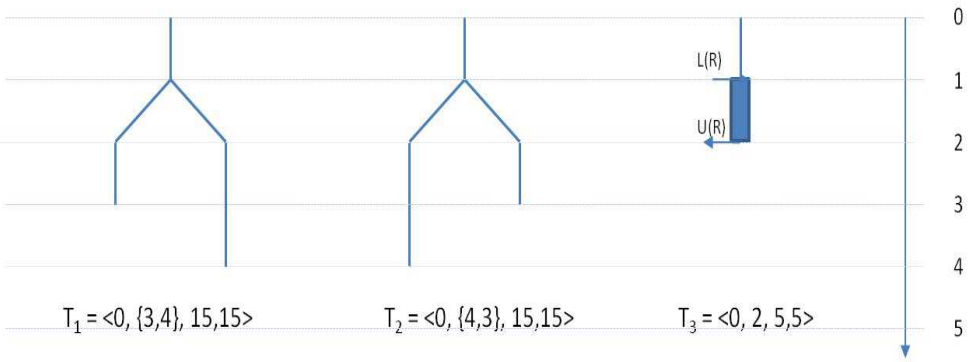


Figure 3.11 – Représentation graphique des tâches de l'application S

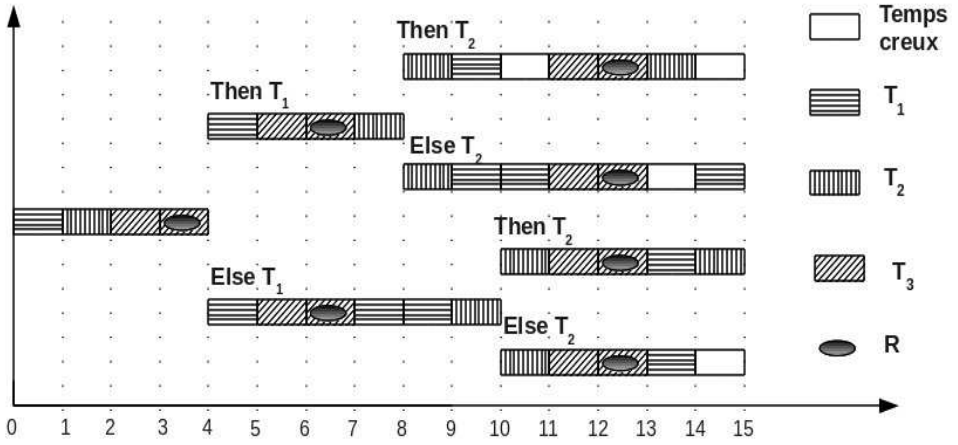


Figure 3.12 – Un arbre d'ordonnancement du système S

3.4.4 Synthèse de l'approche

L'approche globale d'analyse de **Pailler et Al.** est illustrée figure 3.13.

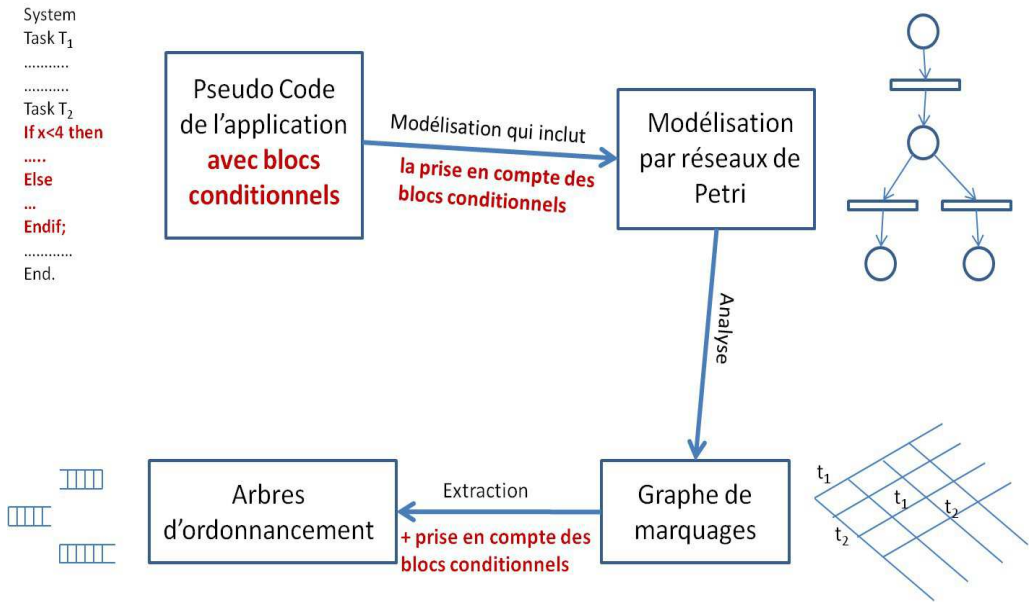


Figure 3.13 – Illustration de la méthodologie d'analyse des applications dans l'approche **Pailler et Al.**

Le point d'entrée de l'approche est le pseudo-code de l'application. Ce pseudo-code intègre de façon explicite les instructions conditionnelles.
L'analyse de l'application se fait en quatre étapes :

1. repérer les blocs conditionnels afin de les considérer pendant la modélisation. Il n'y a plus une pire durée d'exécution C pour une tâche, mais un multi-ensemble de durées, chacune correspondant au choix d'une branche de la tâche ;
2. modéliser l'application comme un réseau de Petri avec marquages terminaux fonctionnant sous la règle du tir maximal, et qui considère de façon explicite les blocs conditionnels ;
3. construire le graphe de marquages du réseau de Petri ;
4. extraire de ce graphe de marquages soit un ensemble de séquences valides, chaque séquence correspondant au choix d'une branche conditionnelle, soit un ensemble d'arbres d'ordonnancement, un arbre d'ordonnancement étant un ordonnancement valide indépendamment de la branche conditionnelle choisie.

3.4.5 Enrichissement de l'approche

Pour les tâches appartenant à notre contexte d'étude, les travaux de **Pailler et Al.** constituent une bonne base pour l'analyse des systèmes.

Toutefois, leur analyse ne prend en compte ni les relations **intra-tâches**, ni les relations **inter-tâches**.
Nous nous proposons de traiter ces points dans notre approche.

Deuxième partie

Contribution

Chapitre 4

Modélisation d'une tâche

"*Quand une science est à bout d'arguments, elle élargit son vocabulaire*"
Jacques Deval

Sommaire

Nous présentons dans ce chapitre le modèle de tâche que nous avons retenu pour répondre à notre problématique : prendre explicitement en compte les instructions conditionnelles contenues dans le corps d'une tâche, et les éventuelles interactions pouvant exister (relations intra-tâches).
Nous commençons par donner la structure des tâches que nous manipulons, ensuite nous introduisons les représentations arborescentes et linéaires.
Puis nous justifions le choix de l'approche arborescente.
Enfin, nous présentons les modèles d'exécution et temporel des tâches.

4.1 Modèle structurel d'une tâche

Les tâches considérées dans cette thèse sont constituées d'un ensemble d'instructions séquentielles exécutées sur apparition d'un même évènement ou d'un même ensemble d'évènements.
Nous donnons une définition un peu plus formelle de la notion de tâches utilisée.

Définition 4.1.1 Une tâche
Une **tâche** est une séquence finie composée de blocs généraux B_i.
Chaque bloc B_i peut être :

1. un bloc d'instructions non conditionnelles de durée d_i qu'on notera $\mathrm{b}(d_i)$;

2. une primitive temps-réel de durée supposée nulle.
   Nous rappelons en effet que nous avons supposé que les durées des primitives temps-réel étaient contenues dans les durées des blocs adjacents.
   Ces primitives sont :
   (a) envoi d'un message M qui sera noté $S(M)$ et qui signifie que la tâche dépose un message M dans une boite à lettre implicite ;
   (b) réception d'un message M qui sera notée $R(M)$ et qui signifie qu'un message est récupéré d'une boite à lettre implicite ;
   (c) prise d'une ressource R qui sera notée $L(R)$ et qui signifie qu'un verrou est placé sur la ressource R ;
   (d) libération d'une ressource R qui sera notée $U(R)$ et qui signifie qu'un verrou est levé de la ressource R ;
3. un bloc conditionnel : *If* $Test_i$ *Then* SB_i^+ *Else* SB_i^- *EndIf*, où SB_i^+ et SB_i^- sont des séquences de blocs généraux et $Test_i$ désigne un test de durée t_{d_i}.

Ainsi, une tâche T sera décrite dans son approche structurelle par :
$T = B_1.....B_n$, où B_i est un bloc général décrit par la définition 4.1.1, et n est le nombre de blocs généraux.

Nous considérons l'exemple 1.2.2.1 du système des essuie-glaces, afin de donner la structure des tâches le constituant.

Exemple 4.1.1 Structure des tâches du système des essuie-glaces
L'analyse des spécifications de l'exemple 1.2.2.1 nous conduit à modéliser le système de contrôle des essuie-glaces par trois (3) tâches dépendantes :

1. la tâche $T_{Reception}$ qui lit l'état du *comodo* et envoie les informations à la tâche T_{Calcul} (par envoi de message) ;
2. T_{Calcul} récupère ces informations, prend en compte la vitesse du véhicule et la quantité d'eau fournie par le capteur d'eau sur le pare-brise et donne les directives à la tâche $T_{Coordonne}$ (par envoi de message) ;
3. $T_{Coordonne}$ coordonne le balayage du pare-brise par les essuie-glaces (par échange de messages).

Nous donnons en annexe .1 la structure des tâches des essuie-glaces.

Nous définissons la notion de tâche correcte.

Définition 4.1.2 Correction d'une tâche
Nous dirons qu'une tâche est **correcte** (et donc prête à l'étude) si :

1. toutes les ressources prises sont libérées par la suite ;
2. il n'existe pas de libération de ressources non prises préalablement.

La tâche sera dite **non correcte** si l'un des points 1 ou 2 n'est pas respecté.

Par la suite, sauf précision explicite, nous ne considérerons que des tâches correctes.
Notre objectif en effet n'est pas de faire de la validation structurelle du code des tâches, mais de réaliser la validation temporelle d'un ensemble de tâches correctement implémentées.

Nous proposons maintenant une représentation arborescente des tâches, qui met en évidence sa structure, et partant les instructions conditionnelles.

4.1.1 Modèle arborescent

Nous pouvons visualiser la tâche sous une forme arborescente qui représente sa structure.
Nous adoptons les conventions suivantes :

1. un lien de l'arbre modélise l'exécution d'une unité de temps processeur ;
2. un bloc de durée d est représenté par d liens comme illustré figure 4.1 ;

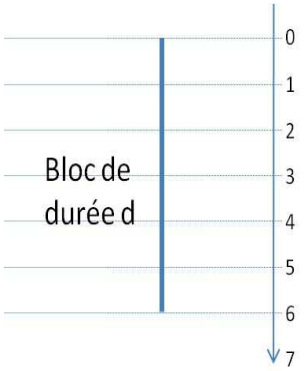


Figure 4.1 – Représentation d'un bloc de durée 6

3. les représentations des primitives temps-réel sont données figure 4.2 ;

Figure 4.2 – Représentation des primitives temps-réel

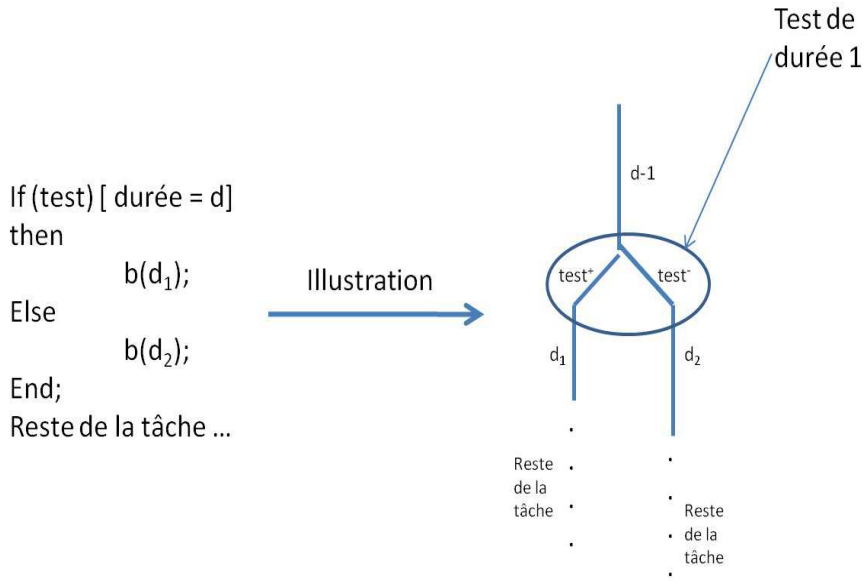


Figure 4.3 – Représentation d'un bloc conditionnel

4. nous illustrons figure 4.3 la représentation des instructions conditionnelles.
   Nous considérons ici, et nous conserverons cette approche dans la suite, qu'un test de durée d est en fait un bloc de durée $d-1$ suivi d'un test de durée 1.
   Ceci correspond intuitivement à l'idée qu'il faut attendre la fin du test pour décider de la branche à suivre.
   Enfin, il faut représenter la tâche dans sa globalité, c'est à dire le séquencement des instructions.
   Pour mettre en évidence le fait que les instructions restantes seront exécutées quelque soit la branche choisie, nous recopions à la fin de chacune des branches *then* et *else* l'ensemble des instructions restant à exécuter.

Une illustration complète de la représentation d'une tâche est donnée figure 4.4.
La figure 4.4 se décrit ainsi : après 2 unités de temps processeur, la tâche représentée envoie un message M, puis s'exécute pendant 1 unité de temps processeur.
Ensuite, elle prend une ressource R, qu'elle ne libérera qu'après un certain temps : si la composante positive du test est choisie, la ressource sera libérée après 5 unités de temps processeur, sinon, ce sera après 7 unités de temps.
Et dans tous les cas, la tâche s'exécutera encore pendant 2 unités de temps processeur.
Nous pouvons remarquer la gestion du test de durée 3, qui consiste à supposer que la tâche s'exécute d'abord pendant 2 unités de temps, et que le test s'exécute pendant 1 unité.

Nous donnons figure 4.5 la représentation arborescente des tâches constituant le système des essuie-glaces.

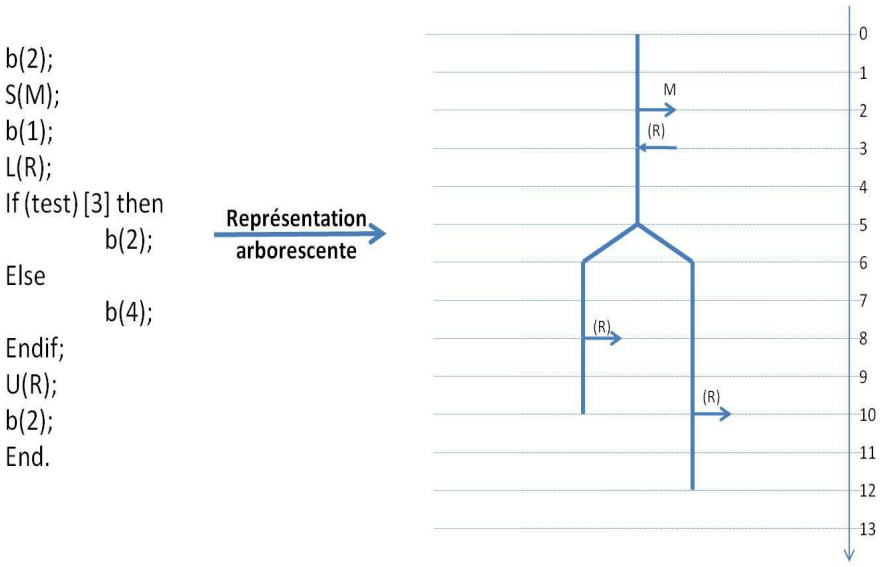


Figure 4.4 – Représentation d'une tâche

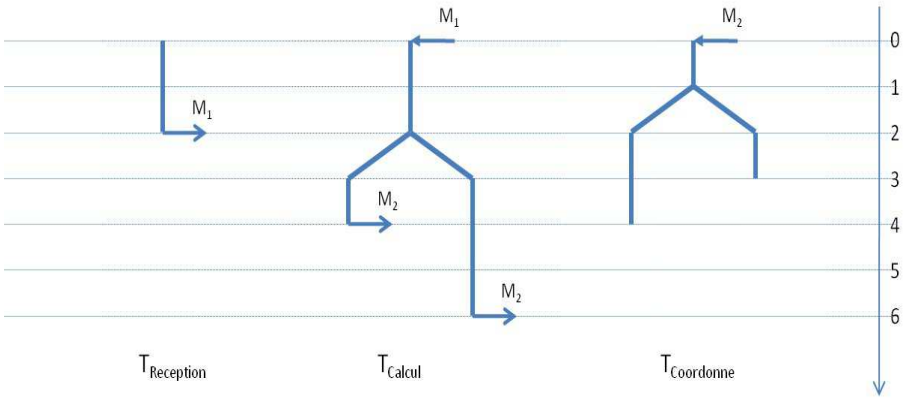


Figure 4.5 – Représentation arborescente des tâches du système de contrôle des essuie-glaces

4.1.2 De l'arborescent vers le linéaire

De façon générale, l'objectif recherché pendant l'étude des applications temps-réel est la garantie absolue de sûreté.
Pour cela, les auteurs préfèrent travailler dans le pire cas. C'est la raison pour laquelle les applications classiques sont représentées à l'aide de modèles linéaires qui intègrent toutes les contraintes de tous les comportements.
Ces modèles consistent à représenter une tâche comme une séquence linéaire de blocs d'instructions entrecoupés de primitives temps-réel.
Dans cette approche, en présence d'un bloc conditionnel, la séquence représentée n'est plus le test conditionnel avec chacune de ses branches *then* et *else*, mais la plus longue branche [Wilhelm 2008].
Quand les blocs conditionnels contiennent des primitives temps-réel, nous n'avons pas trouvé dans la littérature de méthodologie précise expliquant le passage du modèle arborescent au modèle linéaire.
Nous proposons dans cette section une approche de prise en compte des blocs conditionnels qui contiennent des primitives temps-réel pendant le passage du modèle arborescent au modèle linéaire.

Le problème de la linéarisation

Dans cette section, l'objectif est de définir la fonction *Lineaire* :
$Lineaire$: *Ensemble de blocs généraux* $\bigcup$ *Ensemble des tâches* $\rightarrow$ *Ensemble de blocs linéaires* $\bigcup$ *Ensemble des tâches* telle que, pour une tâche $T = B_1.....B_n$, où B_i est un bloc général de la définition 4.1.1,
$Lineaire(T) = Lineaire(B_1).Lineaire(B_2).....Lineaire(B_n)$:

1. si B_i est un bloc d'instructions non conditionnelles, alors $Lineaire(B_i) = B_i$;
2. si B_i est une primitive temps-réel, alors $Lineaire(B_i) = B_i$;
3. si B_i est un bloc conditionnel, c'est à dire
   $B_i = if\ Test\ then\ SB^+\ else\ SB^-\ endif$, où SB^+ (SB^-) est la séquence de blocs généraux qui correspond au choix de la composante positive (négative) de $Test$, alors, la littérature recommande de choisir la plus longue branche [Wilhelm 2008], et nous n'avons rien trouvé quant au placement exact des éventuelles primitives temps-réel.
   Nous allons proposer une solution à ce problème.

Notre problème est le positionnement des primitives temps-réel sur l'unique longue branche de la tâche dans sa représentation linéaire, sans que l'application ne perde ses propriétés initiales, comme par exemple le fait qu'une application initialement ordonnançable reste ordonnançable après sa transformation.
Nous allons illustrer les problèmes rencontrés, et les solutions proposées sur des exemples, afin de déduire les règles de transformation.

4.1.2.1 Gestion de l'envoi des messages

Considérons une application temps-réel S constituée de 2 tâches dépendantes T_1 et T_2 dont les représentations arborescentes sont données figure 4.6.

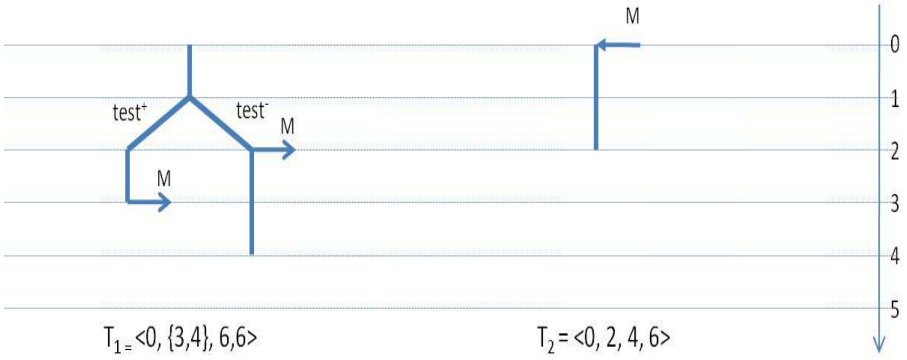


Figure 4.6 – Système utilisé pour étudier le positionnement des messages envoyés

La tâche T_2 n'a pas d'instruction conditionnelle.
Par conséquent, sa représentation linéaire est équivalente à sa représentation arborescente.
La tâche T_1 a une instruction conditionnelle.
Pour sa représentation linéaire, nous considérons, comme le recommande l'état de l'art, la plus longue branche, qui a pour pire durée d'exécution 4 unités de temps.
Le problème qui se pose est le placement de l'instruction d'envoi de message $S(M)$:

1. si on garde uniquement ce qui se passe dans la plus longue branche, c'est à dire si ce message est envoyé uniquement à l'instant $t = 2$ (au plus tôt), alors, les représentations linéaires des tâches T_1 et T_2, et une séquence d'ordonnancement valide du système sont données figure 4.7.
   Le système linéarisé est donc ordonnançable.
   Par contre, si on garde la représentation arborescente, il n'existe pas d'arbre d'ordonnancement valide.
   En effet, S n'est pas localement ordonnançable.
   Le choix dans T_1 de la branche correspondant à $test^+$ fait en sorte que le message M soit envoyé après 3 unités de temps d'exécution.
   La tâche T_2 va ainsi rater son échéance.
   En conclusion, ce cas ne doit pas être considéré pour la linéarisation, parce que la cohérence d'ordonnançabilité n'est pas vérifiée, la linéarisation ayant pour but de renforcer la sécurité.
2. si nous considérons que ce message est envoyé aux instants $t = 2$ et $t = 3$ (envois multiples), c'est à dire que nous considérons que les messages émis dans les différentes branches sont des messages distincts, alors il faut modifier T_2, pour avoir autant de réceptions.

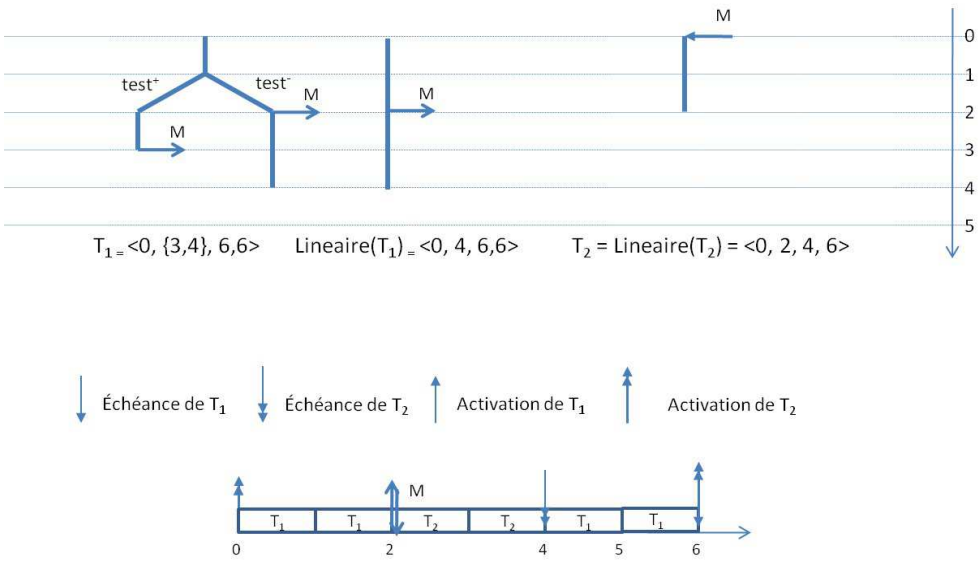


Figure 4.7 – Système linéaire de S, avec primitive au plus tôt, et une séquence d'ordonnancement valide du système linéarisé

Ce qui n'est pas souhaitable, car la linéarisation doit dépendre uniquement de la tâche concernée ;

3. la dernière option, et c'est celle que nous retiendrons, consiste à considérer l'envoi du message M à l'instant $t = 3$ (au plus tard) dans le modèle linéaire. Remarquons que dans ce cas, le système n'est pas ordonnançable quelque soit l'approche considérée.

Nous énonçons la règle 4.1.1 pour la prise en compte des envois de messages dans les représentations linéaires des blocs conditionnels.

Règle 4.1.1 Envoi au plus tard d'un message
Soit un bloc conditionnel $B = if\ test\ then\ SB^+\ else\ SB^-\ endif$ où SB^+ et SB^- sont des séquences de blocs généraux.
Si $\exists\ S(M)$ qui apparait dans SB^+ à un instant t_1, et qui apparait dans SB^- à un instant t_2, alors
$Lineaire(B) = FusionMax(b(d).Lineaire(SB^+),\ b(d).Lineaire(SB^-))$ où $b(d)$ est un bloc de durée d (dans la fusion, on perd l'identité des tests), et l'opérateur $FusionMax$ est défini par :
$FusionMax$: *Ensemble de blocs linéaires X Ensemble de blocs linéaires* $\rightarrow$ *Ensemble de blocs linéaires* tel que :
$FusionMax(B, B')$ = *le plus long entre* B *et* B' *dans lequel on a rajouté la primitive d'envoi de message à l'instant* $t = max(t_1, t_2)$.

Nous montrerons dans le chapitre 7 que la règle 4.1.1 garantit qu'après la transformation du modèle arborescent, les conclusions d'ordonnançabilité sont préservées, c'est à dire que si le modèle linéaire est ordonnançable, le modèle arborescent l'est

également.
Remarquons que la règle 4.1.1 peut être justifiée du fait que nous avons supposé que l'envoi d'un message n'était pas bloquant.
On peut donc se permettre de retarder l'envoi au plus tard.

4.1.2.2 Gestion de la réception des messages

Considérons l'application S constituée de 3 tâches dépendantes données figure 4.8.

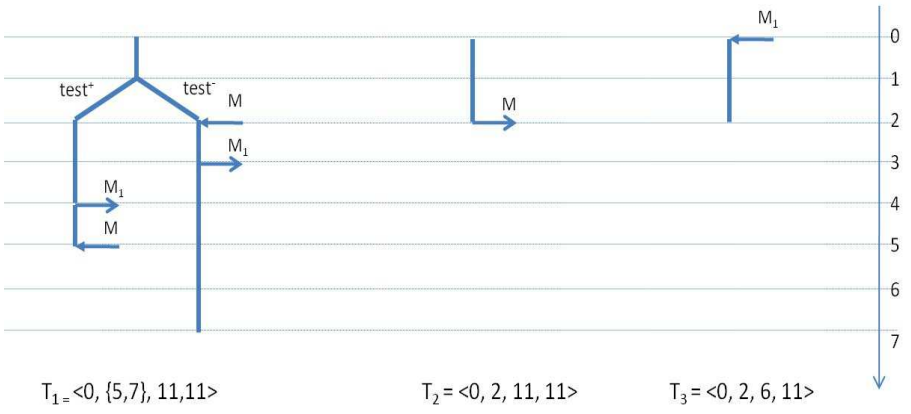


Figure 4.8 – Système utilisé pour étudier le positionnement des messages reçus

Les tâches T_2 et T_3 n'ont pas d'instructions conditionnelles.
Par conséquent, leurs représentations linéaire et arborescente sont semblables.
La tâche T_1 a une instruction conditionnelle.
Pour sa représentation linéaire, nous considérons donc la plus longue branche, qui a pour pire durée d'exécution 7 unités de temps.
Nous avons déjà donné règle 4.1.1 une méthode pour placer l'instruction d'envoi de message $S(M_1)$.
Pour le cas de $Lineaire(T_1)$, $S(M_1)$ sera placé à l'instant 4 (le plus tard).
Le problème qui se pose est le placement de l'instruction de réception de message $R(M)$:

1. le cas où dans la linéarisation on suppose que le message est reçu aux instants 2 et 5 (réceptions multiples) comme deux messages distincts est à exclure.
   En effet, cela amènerait à modifier la tâche T_2, ce qui n'est pas souhaitable ;
2. si nous supposons que dans la linéarisation le message est uniquement reçu à l'instant 5 (au plus tard), alors, les représentations linéaires des tâches T_1, T_2 et T_3, et une séquence d'ordonnancement valide du système sont données figure 4.9.
   Le système linéarisé est donc ordonnançable.
   Par contre, si on garde la représentation arborescente, il n'existe pas d'arbre

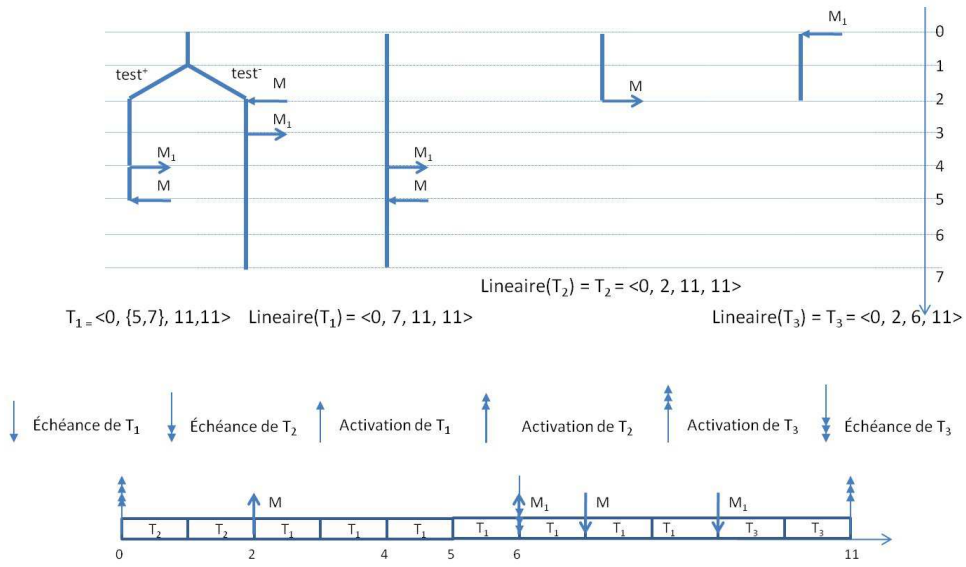


Figure 4.9 – Système linéaire de S, avec réception au plus tard, et une séquence d'ordonnancement valide du système linéarisé

d'ordonnancement valide.
En effet, S n'est pas localement ordonnançable.
Le choix dans T_1 de la branche correspondant à $test^-$ nous oblige, pour qu'il y ait réception du message M, d'attendre 2 unités de temps pour l'envoi du message M par T_2, ensuite il faut attendre 3 unités de temps pour que le message M_1 soit envoyé.
La tâche T_3 va rater son échéance.
En conclusion, comme précédemment, ce cas ne doit pas être considéré pour la linéarisation ;

3. la dernière option, et c'est celle que nous retiendrons, consiste à considérer la réception du message M à l'instant $t = 2$ (au plus tôt) dans le modèle linéaire. Remarquons que dans ce cas, le système n'est pas ordonnançable quelque soit l'approche considérée.

Nous énonçons la règle 4.1.2 pour la prise en compte des réceptions de messages dans les représentations linéaires des blocs conditionnels.

Règle 4.1.2 Réception au plus tôt d'un message
Soit un bloc conditionnel $B = if\ test\ then\ SB^+\ else\ SB^-\ endif$ où SB^+ et SB^- sont des séquences de blocs généraux.
Si $\exists\ R(M)$ qui apparait dans SB^+ à un instant t_1, et qui apparait dans SB^- à un instant t_2, alors
$Lineaire(B) = FusionMin(b(d).Lineaire(SB^+),\ b(d).Lineaire(SB^-))$ où $b(d)$ est un bloc de durée d (dans la fusion, on perd l'identité des tests), et l'opérateur $FusionMin$ est défini par :

FusionMin : *Ensemble de blocs linéaires X Ensemble de blocs linéaires* → *Ensemble de blocs linéaires* tel que :
$FusionMin(B, B') =$ *le plus long entre* B *et* B' *dans lequel on a rajouté la primitive de réception de message à l'instant* $t = min(t_1, t_2)$.

La règle 4.1.2 garantit qu'après la transformation du modèle arborescent, les conclusions d'ordonnançabilité sont respectées, ou du moins elles ne sont pas violées du fait d'une représentation mal structurée.
Remarquons que la règle 4.1.2 peut être justifiée du fait que nous avons supposé que la réception des messages était bloquante.
On ne peut donc pas, dans le cas où la solution consiste à ne pas considérer plusieurs réceptions distinctes, se permettre de retarder la réception au plus tard.
Il est préférable de recevoir plus tôt, afin de permettre au système d'*évoluer*.

4.1.2.3 Gestion des ressources

Nous nous intéressons à la prise en compte des ressources dans la représentation linéaire.
Comme dans les cas de l'envoi de messages et de la réception de messages, nous illustrons le problème sur un exemple, afin de donner une règle générale de prise en compte de ressources.
Considérons une tâche conditionnelle T qui utilise 2 ressources R et R' durant son exécution comme illustré figure 4.10 (a).
Pour prendre en compte l'utilisation des ressources, nous proposons de faire l'union des sections critiques.
Nous aurons besoin des ensembles $Util(R)$ et $\overline{Util(R)}$ pour décrire de façon précise cette union.
Soit donc R une ressource.
$Util(R) = \{(d, l)$ / *sur l'une des branches,* R *est prise à l'instant* d*, et libérée à l'instant* $l\}$
Nous supposons que les éléments de $Util(R)$ sont classés par ordre croissant de d.
Nous construisons $\overline{Util(R)}$:

1. *répéter*
   (a) *si* $\exists\ (d, l)$ *et* (d', l') *dans* $Util(R)$ *tel que* $d \leq d' \leq l$ *et* $(d, l) \neq (d', l')$, *alors*
      i. *supprimer* (d, l) *et* (d', l') *de* $Util(R)$;
      ii. *ajouter* $(d, max(l, l'))$ *à* $Util(R)$;
2. *jusqu'à ce que tous les intervalles de* $Util(R)$ *soient deux à deux disjoints.*

Nous notons $\overline{Util(R)}$ l'ensemble final obtenu.
Nous utilisons la règle 4.1.3 pour positionner les instructions de prise et de libération de ressources sur la représentation linéaire d'une tâche.

Règle 4.1.3 Prise de la ressource au plus tôt et libération au plus tard
Soit une tâche T qui utilise des ressources.
Pour obtenir le positionnement des ressources sur sa représentation linéaire :

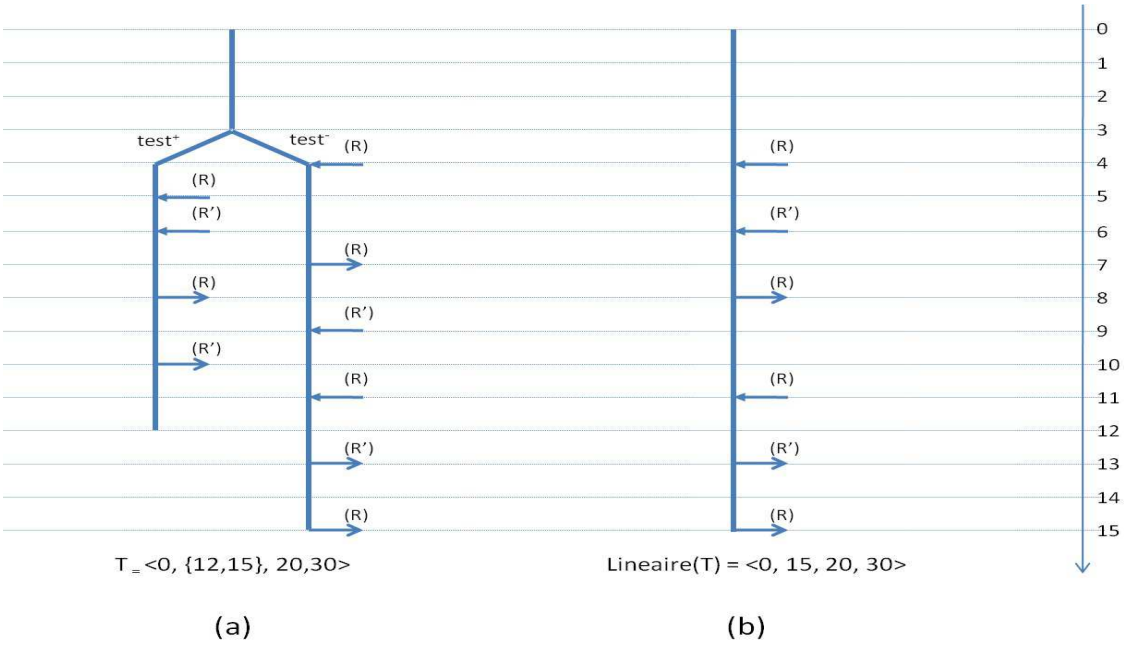


Figure 4.10 – (a) Tâche T utilisée pour expliquer la prise en compte des ressources lors du passage de la représentation arborescente à la représentation linéaire (b) Représentation linéaire de la tâche T

1. *on considère la plus longue branche ;*
2. *pour chaque ressource R de la tâche :*
   (a) *on calcule $\overline{Util(R)}$;*
   (b) *on place $L(R)$ à chaque instant d tel que $(d, l) \in \overline{Util(R)}$;*
   (c) *on place $U(R)$ à chaque instant l tel que $(d, l) \in \overline{Util(R)}$.*

Pour revenir à l'exemple de la figure 4.10 (a), les ensembles $Util(R)$ et $Util(R')$ sont :
$Util(R) = \{(4, 7),\ (5, 8),\ (11, 15)\}$ et $Util(R') = \{(6, 10),\ (9, 13)\}$.
De même, les ensembles $\overline{Util(R)}$ et $\overline{Util(R')}$ obtenus sont :
$\overline{Util(R)} = \{(4, 8),\ (11, 15)\}$ et $\overline{Util(R')} = \{(6, 13)\}$.

En appliquant la règle 4.1.3, nous donnons figure 4.10 (b) la représentation linéaire de la tâche T.

La transformation d'un modèle arborescent vers un modèle linéaire se fera en utilisant les règles 4.1.1, 4.1.2 et 4.1.3.

Remarque 4.1.1 Gestion de conflits entre les ressources et les messages
Si à un même instant, on prend des ressources, et on attend ou envoie des messages, les règles de correction sont :

1. les primitives temps-réel de communication précèdent les primitives temps-réel de prise de ressources ;
2. les libérations de ressources précèdent les attentes de messages.

Une illustration de la transformation d'une représentation arborescente en une représentation linéaire est donnée figure 4.11.

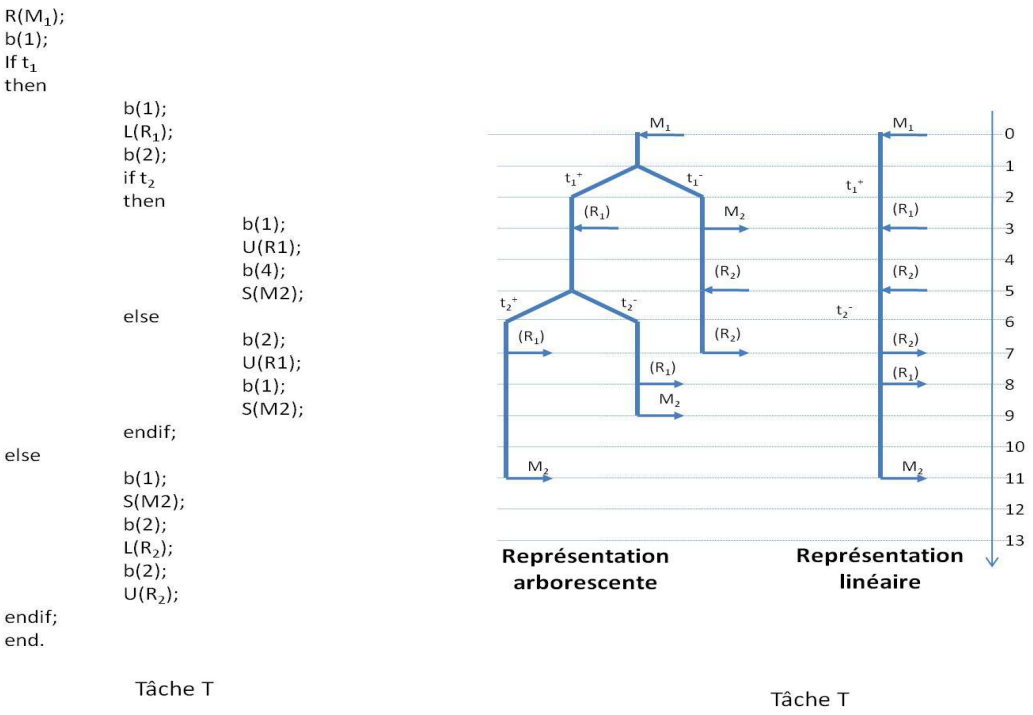


Figure 4.11 – Passage de la représentation arborescente à la représentation linéaire

Nous commençons par considérer la branche la plus longue correspondant au test t_2. Il s'agit de la composante positive t_2^+.
En appliquant la règle 4.1.1, l'envoi du message M_2 se fait à l'instant 11.
Ensuite nous remontons au test t_1.
La branche la plus longue correspond à la composante t_1^+.
En appliquant de nouveau la règle 4.1.1, nous conservons l'envoi du message M_2 à l'instant 11.
La réception du message M_1 à l'instant 0 ne change pas.
Pour la gestion des ressources, nous avons :
$Util(R_1) = \{(3,7),\ (3,8)\}$ et $\overline{Util(R_1)} = \{(3,8)\}$
De même $Util(R_2) = \{(5,7)\}$ et $\overline{Util(R_2)} = \{(5,7)\}$
La représentation obtenue figure 4.11 s'obtient donc en plaçant $L(R_1)$ à l'instant 3, $U(R_1)$ à l'instant 8, $L(R_2)$ à l'instant 5 et $U(R_2)$ à l'instant 7.

Nous donnons figure 4.12 les représentations linéaires des tâches du système des essuie-glaces.

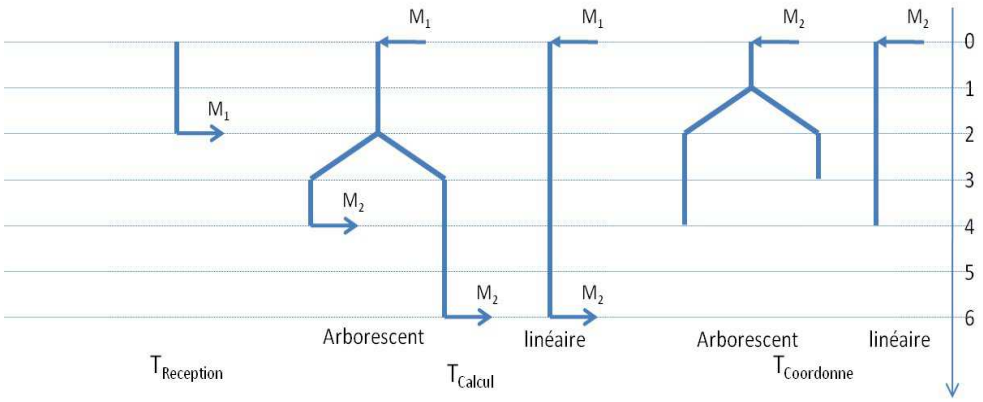


Figure 4.12 – Représentation classique linéaire des tâches du système de contrôle commande des essuie-glaces

Les explications pour comprendre ces représentations linéaires sont similaires à celles données pour la figure 4.11.

La représentation classique linéaire ne considérant pas explicitement toutes les branches des blocs conditionnels, elle ne peut être adaptée si l'objectif est la gestion détaillée de ces blocs.
Une autre raison qui nous pousse à considérer le modèle arborescent est la *sur réservation* des ressources dans le modèle linéaire.

L'exemple de la figure 4.11 illustre bien cette sur réservation : la tâche T utilise à chaque période soit la ressource R_1, soit la ressource R_2. On voit pourtant que le modèle linéaire de T lui réserve et R_1, et R_2.

Par la suite, sauf mention explicite, la représentation considérée sera la représentation arborescente des tâches.

4.2 Modèle d'exécution

Il s'agit ici d'examiner les comportements de la tâche pendant son exécution. Deux approches sont envisagées et nous permettent de tenir compte du caractère arborescent des tâches :

1. le modèle de chemins ;
2. le modèle d'arbre.

Nous commençons par donner quelques notations que nous allons utiliser.

4.2.1 Notations

Les tâches d'une application seront notées $T_1 \dots T_n$, où n est le nombre de tâches.

Dans le cas de notre exemple des essuie-glaces (annexe .1), nous noterons :

1. $T_{Reception}$ par T_1 ;
2. T_{Calcul} par T_2 ;
3. $T_{Coordonne}$ par T_3.

Dans la suite :

1. $\mathcal{R}_i$ désigne l'ensemble des ressources utilisées par la tâche T_i ;
2. $\mathcal{EM}_i$ est l'ensemble des messages émis par la tâche T_i ;
3. $\mathcal{RM}_i$ est l'ensemble des messages reçus par la tâche T_i ;
4. $\mathcal{T}est_i$ désigne l'ensemble des tests présents dans le code de T_i. Nous notons $k_i = |\mathcal{T}est_i|$, le nombre de ces tests ;
5. a_i est une instruction élémentaire unitaire de la tâche T_i.
   Nous avons en effet supposé que les actions élémentaires sont de durées 1.
   $b(d)$ est donc une séquence de d actions élémentaires ;
6. $\mathcal{R}esTest_i = \{t_{ij}^+,\ t_{ij}^-\ /\ t_{ij} \in \mathcal{T}est_i\}$ est l'ensemble des résultats des tests.
   Il s'agit des résultats positifs et négatifs des tests présents dans T_i.
   t_{ij}^+ (t_{ij}^-) est le résultat positif (négatif) du test t_{ij} de la tâche T_i.

Ces notations nous permettent de définir l'alphabet d'une tâche.

Définition 4.2.1 Alphabet d'une tâche T_i
Nous définissons l'**alphabet d'une tâche** T_i, $i \in \mathbb{N}$ par :
$\alpha\beta_i = \{a_i\} \cup \{t_{ij}^s \textit{ tel que } t_{ij}^s \in \mathcal{R}esTest_i \textit{ et } s \in \{+,\ -\}\}$
$\cup \{(L(R), U(R) \textit{ tel que } R \in \mathcal{R}_i\}$
$\cup \{S(M) \textit{ tel que } M \in \mathcal{EM}_i\} \cup \{R(M) \textit{ tel que } M \in \mathcal{RM}_i\}$

Il s'agit de l'ensemble des élements qu'on peut retrouver dans le corps d'une tâche.
Cet alphabet va nous permettre de décrire les chemins et l'arbre d'une tâche.

4.2.2 Les chemins

Une tâche est caractérisée par l'ensemble de ses comportements ou de ses chemins.
La notion de chemin isole un comportement de la tâche.
Par la suite les notions de comportement d'une tâche et de chemin d'une tâche seront semblables.

Nous parlons dans un premier temps des comportements potentiels d'une tâche.

À ce stade, les relations intra-tâches ne sont pas encore considérées. La notion de comportement (effectif) intègrera ces relations.

Un comportement potentiel d'une tâche correspond à une branche de son modèle arborescent.
Si nous notons $Comp_i$ l'ensemble des comportements potentiels de la tâche T_i, un comportement potentiel $Comp_{il}$ (le l^{eme} comportement potentiel de la tâche T_i) de $Comp_i$ est un mot de taille finie écrit sur l'alphabet $\alpha\beta_i$.
Par la suite, nous notons C_{il} la durée d'exécution du comportement potentiel $Comp_{il}$.
Nous nous intéressons à l'obtention des différents comportements potentiels d'une tâche donnée.
Nous donnons définition 4.2.2 une définition constructive et formelle de l'ensemble des comportements potentiels d'une tâche.

Définition 4.2.2 Comportements potentiels d'une tâche
Les **comportements potentiels** de la tâche $T = B_1.....B_n$ sont définis par l'algorithme suivant :

1. si B_1 est un bloc simple ou une primitive temps-réel, les comportements potentiels sont de la forme $\mathcal{B}_1.C$, où C est un comportement potentiel de $B_2.....B_n$ et $\mathcal{B}_1$ est le comportement de B_1 qui correspond à l'exécution séquentielle des instructions constituant B_1 ;
2. si B_1 est un bloc conditionnel, l'ensemble des comportements potentiels de T est l'union de deux ensembles de comportements potentiels suivants :
   (a) $\{Test_1^+.\mathcal{B}_1^+.C$, où C est un comportement potentiel de $B_2.....B_n$ et $\mathcal{B}_1^+$ un comportement potentiel de B_1^+ $\}$ est l'ensemble des comportements potentiels provenant du choix de la branche positive de la conditionnelle ;
   (b) $\{Test_1^-.\mathcal{B}_1^-.C$ où C est un comportement potentiel de $B_2.....B_n$ et $\mathcal{B}_1^-$ un comportement potentiel de B_1^- $\}$ est l'ensemble des comportements potentiels provenant du choix de la branche négative de la conditionnelle.

Cette notion de comportement potentiel n'est pas suffisante pour donner une description des exécutions effectives de la tâche.
En effet, elle ne gère pas les relations **intra-tâches**, qui sont, rappelons le, les éventuelles corrélations qui peuvent exister entre les comportements choisis dans des conditionnelles successives d'une même tâche et portant sur une même valeur.
Ainsi, le choix d'un comportement en présence d'un test conditionnel, peut dépendre des choix précédents effectués.
Nous affinons donc la notion de comportement, afin de ne considérer que les comportements effectifs, que nous appellerons comportements dans la suite du document.
Un **comportement d'une tâche** est donc un comportement potentiel qui ne souffre d'aucune incohérence sémantique, c'est à dire pour lequel il n'existe pas parmi ses éléments des tests successifs dont les résultats seraient *incompatibles*.

Dans ce qui suit, nous ne considérons que les tests portant sur des paramètres

des tâches, qui ne sont pas modifiés par les tâches.
Ce sont par exemple, des tests de type seuillage ou identification portant sur des valeurs issues du contexte. Les autres tests peuvent exister, mais ils ne sont pas considérés par la définition des relations d'incompatibilité.

Soit $\mathcal{T}estPar_i$ l'ensemble de ces tests, et
$\mathcal{R}esTestPar_i = \{t^+,\ t^-\ tel\ que\ t \in \mathcal{T}estPar_i\}$ l'ensemble des résultats de ces tests.
Nous définissons la relation d'incompatibilité $\mathcal{RIT}_i$ sur $\mathcal{R}esTestPar_i$.

Définition 4.2.3 Relation d'incompatibilité dans une tâche
La **relation d'incompatibilité** $\mathcal{RIT}_i$ est définie sur $\mathcal{R}esTestPar_i$ par :
si $t, t' \in \mathcal{R}esTestPar_i$, $t\ \mathcal{RIT}_i\ t'$ si et seulement si $t \Rightarrow \overline{t'}$, $\bar{t}$ étant la négation de t et $\bar{\bar{t}} = t$.
Nous notons $\overline{\mathcal{RIT}_i}$ la négation de $\mathcal{RIT}_i$.
$t\ \overline{\mathcal{RIT}_i}\ t'$ si et seulement si t et t' ne sont pas incompatibles.
Notons enfin que la relation d'incompatibilité entre 2 éléments t et t' de $\mathcal{R}esTestPar_i$ peut aussi s'exprimer en utilisant le *et logique*.
Ainsi, $t\ \mathcal{RIT}_i\ t'$ si et seulement si $t \wedge t' = false$.

Remarque 4.2.1 La relation d'incompatibilité $\mathcal{RIT}_i$ est une relation symétrique.

Preuve 4.2.1 $\mathcal{RIT}_i$ est une relation symétrique
Soient $t_1, t_2 \in \mathcal{R}esTestPar_i$ tel que $t_1\ \mathcal{RIT}_i\ t_2$.
Nous voulons montrer que $t_2\ \mathcal{RIT}_i\ t_1$.
Comme $t_1\ \mathcal{RIT}_i\ t_2$, on a $t_1 \Rightarrow \overline{t_2}$. En utilisant la contraposée, on a $\overline{\overline{t_2}} \Rightarrow \overline{t_1}$, soit $t_2 \Rightarrow \overline{t_1}$. C'est à dire $t_2\ \mathcal{RIT}_i\ t_1$. □

Nous allons maintenant définir la notion de comportement d'une tâche.

Définition 4.2.4 Comportement d'une tâche
Un comportement potentiel $Comp_{il}$ d'une tâche T_i est un **comportement (effectif)** si et seulement s'il ne contient pas deux (2) éléments t et t' de $\mathcal{R}esTestPar_i$ tels que $t\ \mathcal{RIT}_i\ t'$.
Nous notons $Compeff_i$ l'ensemble des comportements (effectifs) de T_i.

Pour construire les comportements d'une tâche, nous modifions la définition 4.2.2, afin d'éliminer les comportements non effectifs, c'est à dire les comportements potentiels qui possèdent des tests incompatibles.
Nous associons à chaque comportement $Comp$ l'ensemble $MemTest$ des tests obtenus le long du comportement $Comp$. Nous construisons $Comp$ et $MemTest$ en parallèle. $MemTest$ correspond à la mémoire des choix passés.

Méthode 4.2.1 Comportements d'une tâche
Initialement, $MemTest = \emptyset$.
À l'étape courante, les comportements de $B_k \ldots . B_n$, sachant qu'on a déjà obtenu les résultats $MemTest$ sont obtenus par :

1. si B_k est un bloc simple ou une primitive temps-réel, les comportements sont de la forme $\mathcal{B}_k.C$ où C est un comportement de $B_{k+1}.....B_n$ et $\mathcal{B}_k$ est le comportement de B_k.
   L'ensemble $MemTest$ n'est pas modifié, car le bloc sur lequel nous nous trouvons est un bloc simple sans aucun test conditionnel ;
2. si B_k est un bloc conditionnel de test te, il faut tenir compte des tests présents dans $MemTest$, avant de construire les éventuels comportements :
   (a) si $\exists\ t \in MemTest$ et $\exists\ t' \in MemTest$ tel que $t\ \mathcal{RIT}_i\ te^+$ et $t'\ \mathcal{RIT}_i\ te^-$, alors aucun comportement ne peut être construit à partir de ce test. $MemTest$ n'est pas modifié ;

   Remarque 4.2.2 Remarquons qu'il s'agit là d'un cas impossible, car si t et t' sont tels que $t\ \mathcal{RIT}_i\ te^+$ et $t'\ \mathcal{RIT}_i\ te^-$, alors, ils ne peuvent pas être tous les deux dans $MemTest$, car ils sont incompatibles.
   Or, on n'ajoute dans $MemTest$ que les tests non incompatibles avec tous les tests de $MemTest$.

   Preuve 4.2.2 si $t\ \mathcal{RIT}_i\ te^+$ et $t'\ \mathcal{RIT}_i\ te^-$ alors $t\ \mathcal{RIT}_i\ t'$.
   On a $t\ \mathcal{RIT}_i\ te^+$, donc $t \Rightarrow \overline{te^+}$ et $\overline{te^+} = te^-$.
   D'autre part, $t'\ \mathcal{RIT}_i\ te^-$ est équivalent à $te^-\ \mathcal{RIT}_i\ t'$, car $\mathcal{RIT}_i$ est une relation symétrique. Donc $te^- \Rightarrow \overline{t'}$.
   En combinant les trois équations, on a $t \Rightarrow \overline{t'}$ c'est à dire $t\ \mathcal{RIT}_i\ t'$. □

   (b) sinon si $\exists\ t_0 \in MemTest$ tel que $t_0\ \mathcal{RIT}_i\ te^+$ et $\forall\ t \in MemTest$ $t\ \overline{\mathcal{RIT}_i}\ te^-$, alors les comportements sont de la forme $te^-.\mathcal{B}_1^-.C$ associés à $MemTest^- := MemTest \cup \{te^-\}$, où
   C est un comportement de $B_{k+1}.....B_n$ dans lequel tous les tests sont compatibles avec tous les tests de $MemTest^-$, et $\mathcal{B}_1^-$ un comportement de B_1^- dans lequel tous les tests sont compatibles avec tous les tests de $MemTest^-$;
   (c) sinon si $\exists\ t_0 \in MemTest$ tel que $t_0\ \mathcal{RIT}_i\ te^-$ et $\forall\ t \in MemTest$ $t\ \overline{\mathcal{RIT}_i}\ te^+$, alors les comportements sont de la forme $te^+.B_1^+.C$ associés à $MemTest^+ := MemTest \cup \{te^+\}$, où
   C est un comportement de $B_{k+1}.....B_n$ dans lequel tous les tests sont compatibles avec tous les tests de $MemTest^+$, et $\mathcal{B}_1^+$ un comportement de B_1^+ dans lequel tous les tests sont compatibles avec tous les tests de $MemTest^+$;
   (d) sinon, $\forall\ t \in MemTest$, $t\ \overline{\mathcal{RIT}_i}\ te^+$ et $t\ \overline{\mathcal{RIT}_i}\ te^-$.
   Par conséquent, les comportements sont de la forme
   $te^+.\mathcal{B}_1^+.C^+$ associés à $MemTest^+ := MemTest \cup \{te^+\}$
   ou de la forme
   $te^-.\mathcal{B}_1^-.C^-$ associés à $MemTest^- := MemTest \cup \{te^-\}$,
   C^+ étant un comportement de $B_{k+1}.....B_n$ dans lequel tous les tests sont compatibles avec tous les tests de $MemTest^+$,

C^- étant un comportement de $B_{k+1}.....B_n$ dans lequel tous les tests sont compatibles avec tous les tests de $MemTest^-$,
et $\mathcal{B}_1^-$ un comportement de B_1^- dans lequel tous les tests sont compatibles avec tous les tests de $MemTest^-$
et $\mathcal{B}_1^+$ un comportement de B_1^+ dans lequel tous les tests sont compatibles avec tous les tests de $MemTest^+$.

Les relations intra-tâches sont bien prises en compte.
En effet, lorsqu'un test est rencontré, nous vérifions dans l'historique des résultats de tests déjà rencontrés s'il n'en existe pas un avec lequel l'un de ses résultats (positifs ou négatifs) est en incompatibilité.
Si le composant positif (négatif) est incompatible avec un résultat de test déjà rencontré, tous les comportements qui peuvent s'en déduire sont incohérents, et nous ne les construisons pas (cas 2b) et (cas 2c).
Si enfin aucun composant n'est incompatible avec un résultat de test déjà rencontré, nous construisons l'ensemble des comportements induits des deux composantes (cas 2d).

Dans la suite du document, sauf mention explicite, les comportements considérés seront supposés sémantiquement corrects, c'est à dire qu'il s'agira de comportements effectifs.

Une tâche est ainsi définie par l'ensemble de ses comportements (effectifs).

Nous donnons figure 4.13 les modèles d'exécution des tâches du système de contrôle des essuie-glaces lorsqu'elles sont considérées comme un ensemble de chemins ou de comportements.

$T_1 = (b(2).S(M_1))$; 1 comportement

$T_2 = \left(R(M_1).b(2).t^+_{21}.b(1).S(M_2), R(M_1).b(2).t^-_{21}.b(3).S(M_2)\right)$; 2 comportements

$T_3 = \left(R(M_2).b(1).t^+_{31}.b(2), R(M_2).b(1).t^-_{31}.b(1)\right)$; 2 comportements

Figure 4.13 – Modèle d'exécution des tâches du système de contrôle commande des essuie-glaces : approche chemin

Remarquons que dans les cas présentés figure 4.13, les comportements potentiels des tâches sont tous des comportements effectifs, les tâches n'ayant pas de tests incompatibles parmi leurs éléments.

Pour mettre en évidence la prise en compte des relations intra-tâches en utilisant la méthode 4.2.1, nous considérons la tâche T de la figure 4.14 donnée sous sa forme arborescente.
Les quatre comportements potentiels de T sont :

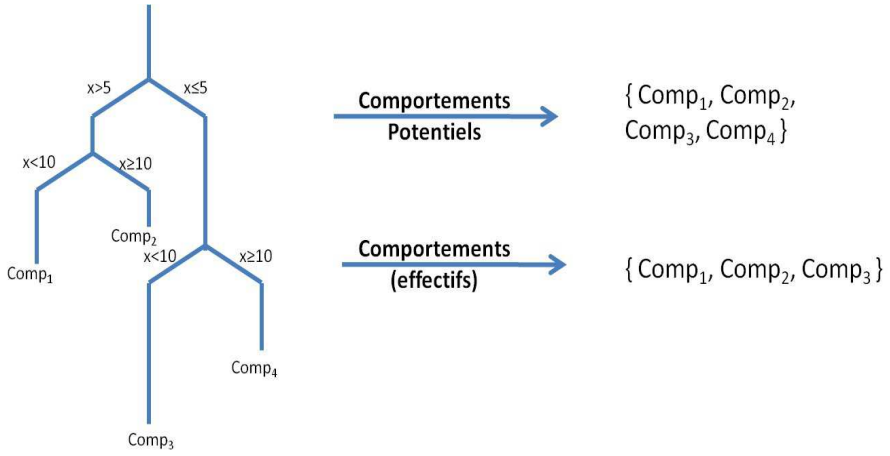


Figure 4.14 – Prise en compte des relations intra-tâches

1. $Comp_1$ induit par les résultats des tests $x > 5$ et $x < 10$;
2. $Comp_2$ induit par les résultats des tests $x > 5$ et $x \geq 10$;
3. $Comp_3$ induit par les résultats des tests $x \leq 5$ et $x < 10$;
4. $Comp_4$ induit par les résultats des tests $x \leq 5$ et $x \geq 10$.

Les tests $x \leq 5$ et $x \geq 10$ sont incompatibles car $x \leq 5 \Rightarrow \overline{x \geq 10}$.
Par conséquent, le comportement potentiel $Comp_4$ est incohérent, et ne fait pas partie des comportements de la tâche.

L'approche présentée prend explicitement en compte les relations intra-tâches, et donne une vue globale de l'ensemble des comportements de l'application à étudier.

Afin de représenter l'ensemble des comportements, nous allons reprendre la structure d'arbre, que nous allons *élaguer*, en éliminant les comportements incohérents.

4.2.3 Les arbres

Nous donnons une définition générale d'un arbre d'exécution.

Définition 4.2.5 Arbre d'une tâche
L'**arbre** d'une tâche est un arbre dont chaque branche correspond à un comportement potentiel de la tâche.

Tout comme pour les comportements d'une tâche, nous définissons la notion d'arbre d'exécution d'une tâche, qui permet de prendre en compte les relations intra-tâches.

Définition 4.2.6 Arbre d'exécution d'une tâche
Un **arbre d'exécution** d'une tâche est un arbre dont nous n'avons gardé que les branches correspondant aux comportements (effectifs).

Nous donnons figure 4.15 l'arbre d'exécution de la tâche T de la figure 4.14.

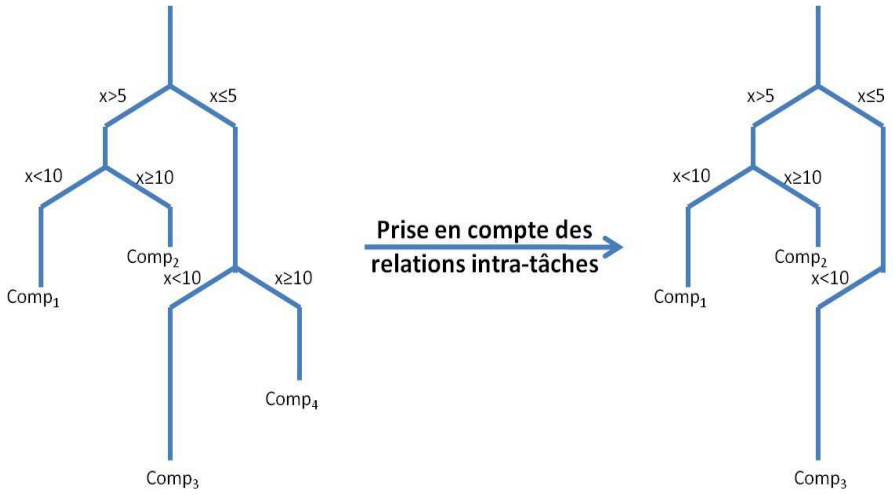


Figure 4.15 – L'arbre et l'arbre d'exécution de la tâche T

Nous pouvons remarquer l'absence du comportement $Comp_4$ incohérent de l'arbre d'exécution de T.

Le modèle d'exécution d'une tâche sous la forme d'un arbre s'obtient en étiquetant les liens de la tâche T_i par l'alphabet $\alpha\beta_i$:

1. les blocs élémentaires $b(d)$ sont décomposés en d blocs élémentaires et unitaires. Chaque bloc unitaire est étiqueté par a_i ;
2. les primitives temps-réel sont étiquetées par la primitive correspondante ;
3. si nous avons un nœud double, les deux liens issus de ce nœud sont étiquetés par t_{ij}^{+} et t_{ij}^{-}, où t_{ij} est un test de $\mathcal{T}est_i$.

Afin de donner une définition complète et formelle des arbres des tâches, nous allons prendre en compte le temps.
Nous associons à chaque nœud η de l'arbre un entier qui correspond à la durée d'exécution de la portion de code qui a conduit à l'état associé.
Cet entier se calcule en utilisant la fonction $Temp$ suivante :

$$\begin{cases} Temp : Ensemble\ des\ nœuds\ de\ l'arbre \rightarrow \mathbb{N}\ telle\ que : \\ Temp(racine) \ = \ 0 \\ Temp(\eta) \ = \ Temp(Pere(\eta)) \ + \ duree(étiquette(\ lien(\ pere(\eta) \rightarrow \eta))) \end{cases}$$

où η est un nœud de l'arbre.
La fonction $Temp$ donne le temps processeur nécessaire pour arriver à l'état d'avancement de la tâche modélisé par un nœud donné.
Dans cette fonction, la fonction *duree* correspond au temps utilisé par la tâche pour

exécuter les différentes actions.
Elle est définie par :

$$\begin{cases} duree : \alpha\beta_i \rightarrow \mathbb{N} \; telle \; que : \\ a_i \longmapsto 1 \\ t_{ij}^+, t_{ij}^- \longmapsto 1 \\ L(R), U(R), S(M), R(M) \longmapsto 0 \end{cases}$$

Cette définition est cohérente avec nos hypothèses :
les tests sont de durées supposées unitaires, les primitives sont de durées supposées nulles, car de durées intégrées dans les blocs adjacents, et l'instruction a_i est une instruction de durée unitaire.

Nous allons maintenant donner une définition formelle des tâches qui intègre la gestion des relations d'incompatibilité.

Définition 4.2.7 Arbre d'exécution d'une tâche
L'**arbre d'exécution** d'une tâche est un arbre dont les nœuds ont pour clé des entiers, et les liens sont étiquetés par l'alphabet $\alpha\beta_i$.
Soit η est un nœud de l'arbre.
Les propriétés suivantes sont vérifiées :

1. si η est la racine de l'arbre, alors $Temp(\eta) = 0$;
2. si $||fils(\eta)|| = 0$, (η une feuille), alors $\exists\, l$ tel que :
   (a) $cle(\eta) = C_{il}$, où C_{il} correspond à la durée d'exécution du l^{eme} comportement de la tâche T_i ;
   (b) le mot étiquetant la branche allant de la racine à η est le l^{eme} comportement de T_i ;
3. si $||fils(\eta)|| = 1$, alors :
   (a) soit $étiquette(lien(\eta, fils(\eta))) \in \{a_i\}$
   $\cup\{L(R), U(R) \; tel \; que \; R \in \mathcal{R}_i\}$
   $\cup\{S(M) \; tel \; que \; M \in \mathcal{EM}_i\} \cup\{R(M) \; tel \; que \; M \in \mathcal{RM}_i\}$
   et $cle(fils(\eta)) = cle(\eta) + duree(étiquette(lien(\eta, fils(\eta))))$;
   (b) soit $\exists\, t_{ij} \in \mathcal{T}est_i$ et $s, s' \in \{+, -\}$ tel que
   $étiquette(lien(\eta, fils(\eta))) = t_{ij}^s$ et
   $cle(fils(\eta)) = cle(\eta) + 1$ (on est donc en présence d'un test)
   et $\exists\, \eta' \in ancetre(\eta)$ tel que :
      i. étiquette$(lien(Pere(\eta'), \eta')) = t_{il}^{s'}$;
      ii. $t_{il}^{s'} \; \mathcal{RIT}_i \; \overline{t_{ij}^s}$;
4. si $||fils(\eta)|| = 2$, alors $\exists\, t_{ij}^+$ et $t_{ij}^- \in \mathcal{T}est_i$ tel que
   (a) étiquette$(lien(\eta, fils_1(\eta))) = t_{ij}^+$;
   (b) étiquette$(lien(\eta, fils_2(\eta))) = t_{ij}^-$;

(c) et $\forall\ \eta'$ ancêtre de η,

i. soit $étiquette(lien(\eta', fils(\eta'))) \in \{a_i\}$
$\cup\{L(R), U(R)\ tel\ que\ R \in \mathcal{R}_i\}$
$\cup\{S(M)\ tel\ que\ M \in \mathcal{EM}_i\} \cup\{R(M)\ tel\ que\ M \in \mathcal{RM}_i\}$;

ii. soit $\exists\ t_{ik} \in \mathcal{T}est_i$ et $s, s' \in \{+, -\}$ tel que
$étiquette(lien(\eta', Fils(\eta'))) = t^s_{ik}$ et $t^+_{ij}\ \overline{\mathcal{RIT}_i}\ t^s_{ik}$ et $t^-_{ij}\ \overline{\mathcal{RIT}_i}\ t^s_{ik}$.

Dans ce cas, $cle(fils_i(\eta)) = cle(\eta) + 1$, $i \in \{1, 2\}$.

Le point 2 assure qu'une branche de l'arbre correspond bien à un comportement de la tâche.
Les points 4 et 3 assurent la prise en compte explicite des conditionnelles dans les tâches.
Le point 4 permet de considérer l'ensemble des comportements induits par les composantes positive et négative de la conditionnelle.
Tandis que le point 3b gère le cas où des relations d'incompatibilité existent entre l'étape courante et une étape passée. Il assure que nous ne considérons pas les comportements incohérents.
Enfin, le point 3a permet de considérer l'exécution des instructions sans tests.

Nous donnons figure 4.16 les modèles formels et les représentations graphiques des tâches du système de contrôle des essuie-glaces.

Il ne reste plus qu'à définir le modèle temporel des tâches, afin de caractériser complètement les tâches d'une application.

4.3 Modèle temporel

Le modèle temporel des tâches se déduit du modèle d'exécution.
La durée d'exécution d'une tâche n'est plus modélisée par sa pire durée d'exécution comme dans les approches classiques linéaires, mais par un **multi-ensemble de durées d'exécution**, qui correspondent aux durées des comportements (effectifs) de la tâche.

Dans l'approche comportement (chemin), ces durées correspondent aux C_{il}, où
$C_{il} = \sum_{k=1}^{p} duree(\alpha_k)$, où $Comp_{il} = \alpha_1 . . . . . \alpha_p\ \ /\ \ \alpha_1, \ldots,\ \alpha_p \in \alpha\beta_i$.

Ces durées correspondent aux clés des nœuds feuilles de l'arbre modélisant la tâche.

Les autres paramètres temporels (r_i, D_i et P_i) ne changent pas par rapport au modèle classique, et sont donnés par le concepteur.
Ainsi, une tâche $T_i = < r_i, \zeta_i, D_i, P_i >$ est temporellement caractérisée par :

1. sa date de réveil r_i ;
2. son *multi-ensemble de durées* ζ_i ;

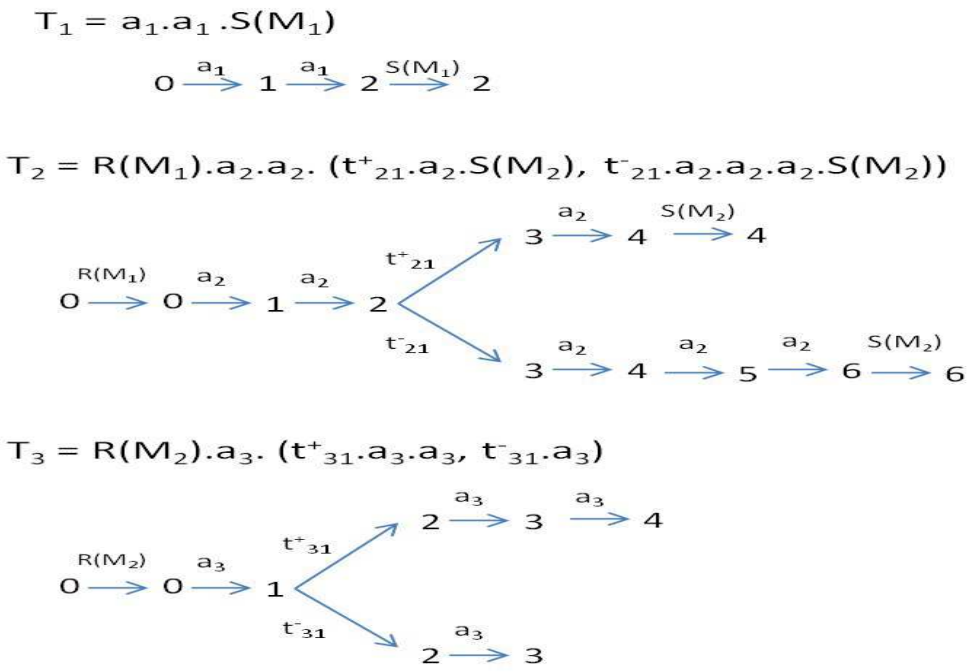


Figure 4.16 – Modèle d'exécution des tâches du système de contrôle commande des essuie-glaces : approche arbre

3. son *délai critique* D_i ;
4. sa *période* P_i.

En considérant de nouveau l'exemple du système de contrôle des essuie-glaces, le modèle temporel des tâches est :

1. $T_1 = < 0, \{2\}, 11, 11 >$;
2. $T_2 = < 0, \{4, 6\}, 11, 11 >$;
3. $T_3 = < 0, \{4, 3\}, 11, 11 >$:

Une tâche est donc complètement caractérisée soit par son arbre d'exécution, soit, ce qui est équivalent, par son ensemble de comportements et par ses paramètres temporels.

Avant de conclure ce chapitre, nous définissons la notion de tâche sémantiquement correcte.

Définition 4.3.1 Tâche sémantiquement correcte
Nous dirons qu'une tâche est **sémantiquement correcte** si et seulement si :

1. elle est correcte (voir définition 4.1.2) ;
2. elle est constituée uniquement de comportement(s) (effectif(s)).

Sans précision explicite, les tâches que nous considérons dans la suite de cette thèse seront sémantiquement correctes.

Bilan

Dans ce chapitre, nous avons proposé trois modèles complémentaires qui permettent de caractériser une tâche temps-réel : les modèles structurel, d'exécution et temporel.
Ces modèles complètent les modèles existants, en permettant de prendre explicitement en compte les instructions conditionnelles contenues dans le code des tâches, ainsi que la sémantique des tests portant sur des variables en entrée non modifiées par la tâche durant son exécution.
L'un de nos objectifs est de modéliser les applications temps-réel.
La modélisation des relations entre les tâches d'une application, en particulier les relations inter-tâches, nécessite de caractériser les applications, et non les tâches prises individuellement.
Ce sera l'objet du chapitre 5.

Chapitre 5

Modélisation de l'application

"*La science se contente de proposer des modèles explicatifs provisoires de la réalité ; et elle est prête à les modifier dès qu'une information nouvelle apporte une contradiction.*"
Albert Jacquard

Sommaire

Après avoir présenté notre modèle de tâche dans le chapitre 4, nous nous intéressons dans celui-ci à la modélisation complète de l'application.
Une application va être définie par :

1. un ensemble de tâches telles que décrites dans le chapitre 4 ;
2. des relations inter-tâches qui permettent d'exprimer les corrélations existant entre les comportements des différentes tâches.

Ce chapitre se divise en trois parties :

1. dans la première partie, nous définissons explicitement la notion de relation inter-tâches utilisée dans cette thèse ;
2. dans la deuxième partie, nous présentons les relations d'ordre structurel. Elles concernent les primitives temps-réel d'envoi/réception de messages et de prise/libération de ressources ;
3. enfin, dans la troisième partie, nous définissons et modélisons la tâche oisive. C'est une tâche fictive qui permet de modéliser l'inactivité processeur. Nous l'intégrons dans notre modélisation afin de pouvoir par la suite décider du placement des temps creux au sein des séquences, et donc de ne pas restreindre les ordonnancements aux seuls ordonnancements conservatifs.

Nous commençons par donner quelques notations.

5.1 Notations

Les notations utilisées dans ce chapitre complètent et étendent les notations du chapitre 4 (section 4.2.1) qui portaient sur une seule tâche, pour prendre en compte toutes les tâches de l'application à étudier.
Une application S est modélisée comme un ensemble de tâches en interaction. Ainsi, une application de n tâches se note $S = \{T_1 \dots T_n\}$.
Les notations suivantes seront utilisées :

1. $\mathcal{T}est = \bigcup_{i=1}^{n} \mathcal{T}est_i$ désigne l'ensemble des tests de l'application ;
2. $\mathcal{R}esTest = \bigcup_{i=1}^{n} \mathcal{R}esTest_i$ est l'ensemble des résultats des tests de l'application ;
3. $\mathcal{T}estPar = \bigcup_{i=1}^{n} \mathcal{T}estPar_i$ désigne l'ensemble des tests de l'application qui portent sur des paramètres en entrée des tâches non modifiables par la tâche elle même ;
4. $\mathcal{R}esTestPar = \bigcup_{i=1}^{n} \mathcal{R}esTestPar_i$ est l'ensemble des résultats de ces tests ;
5. $\mathcal{R} = \bigcup_{i=1}^{n} \mathcal{R}_i$ désigne l'ensemble des ressources de l'application ;
6. $\mathcal{EM} = \bigcup_{i=1}^{n} \mathcal{EM}_i$ est l'ensemble des messages émis dans l'application ;
7. $\mathcal{RM} = \bigcup_{i=1}^{n} \mathcal{RM}_i$ est l'ensemble des messages reçus dans l'application.

Remarquons que nos hypothèses de cohérence structurelle impliquent que $\mathcal{EM} = \mathcal{RM}$.

5.2 Les relations d'incompatibilité

Nous nous intéressons ici aux comportements incompatibles entre plusieurs tâches d'une application. Ces comportements résultent de résultats de tests qui sont incompatibles.
Nous commençons par préciser cette notion, puis nous montrons comment les relations incompatibles sont gérées dans notre étude.

5.2.1 Définition des relations d'incompatibilité

Nous rappelons que les tests considérés pour définir les relations d'incompatibilité portent sur des paramètres en entrée des tâches (les tests de type seuillage ou identification en sont des exemples).
Nous définissons une relation d'incompatibilité entre deux tests de $\mathcal{R}esTestPar$ qui étend la relation définie sur $\mathcal{R}esTestPar_i$.

Définition 5.2.1 Relation d'incompatibilité dans une application
La **relation d'incompatibilité** $\mathcal{RIT}$ est définie sur l'ensemble $\mathcal{R}esTestPar$ par : $\forall\ t,\ t' \in \mathcal{R}estTestPar$, $t\ \mathcal{RIT}\ t'$ si et seulement si :

1. soient $i, j \in 1 \dots n$, n étant le nombre de tâches de l'application, tels que $t \in \mathcal{R}esTestPar_i$ et $t' \in \mathcal{R}esTestPar_j$ alors $r_i = r_j$ et $P_i = P_j$;

2. $t \Rightarrow \overline{t'}$.

Nous notons $t\ \overline{\mathcal{RIT}}\ t'$ la négation de $t\ \mathcal{RIT}\ t'$, et cela signifie que t et t' ne sont pas incompatibles.

La définition 5.2.1 permet de rendre cohérente les mises en relation des tests, instance par instance.
En effet, pour comparer les résultats de deux tests appartenant à deux tâches différentes, il faut s'assurer qu'ils portent sur la même valeur de la variable.
Nos hypothèses (les tâches ont les mêmes dates de réveil et les mêmes périodes) garantissent que les valeurs sont lues par les deux tâches aux mêmes instants, ce qui assure qu'elles sont identiques.
Autrement dit, les valeurs testées sont effectivement les mêmes dans les deux tâches, instance par instance.

Nous illustrons la notion de relation d'incompatibilité sur l'exemple 5.2.1.1.

Exemple 5.2.1.1 Notion de relation d'incompatibilité
Soit l'application temps-réel de la figure 5.1 constituée de 3 tâches T_1, T_2 et T_3.

```
Tâche T1 = <0, {4,5}, 8, 10>
Input Température
{
        si (Température < 4) alors
                b(4);
        sinon
                b(5);
        finsi;
        fin.
}
```

```
Tâche T2 = <0, {5,3}, 8, 10>
Input Température
{
        si (Température > 24) alors
                b(5);
        sinon
                b(3);
        finsi;
        fin.
}
```

```
Tâche T3 = <2, {4,3}, 7, 9>
Input Température
{
        si (Température > 10) alors
                b(4);
        sinon
                b(3);
        finsi;
        fin.
}
```

Figure 5.1 – Système de tâches considéré pour illustrer la notion de relations incompatibles

Nous notons $(Température < 4)_{T_1}$ le résultat du test $Température < 4$ provenant de la tâche T_1.
Nous avons donc $(Température < 4)_{T_1}\ \mathcal{RIT}\ (Température > 24)_{T_2}$.
Par contre, nous avons $(Température < 4)_{T_1}\ \overline{\mathcal{RIT}}\ (Température > 10)_{T_3}$, car les valeurs de la variable $Température$ utilisées dans T_1 et T_3 ne sont pas lues au même moment, donc elles ne sont pas nécessairement les mêmes.

Propriété 5.2.1 La relation d'incompatibilité $\mathcal{RIT}$ est une relation symétrique.

Preuve 5.2.1 $\mathcal{RIT}$ est symétrique
Pour le montrer, nous considérons 2 éléments t_1 et t_2 de $\mathcal{R}estTestPar$ tels que $t_1\ \mathcal{RIT}\ t_2$.
Nous allons montrer que $t_2\ \mathcal{RIT}\ t_1$:

1. soient $i, j \in 1 \dots n$ tels que $t_2 \in \mathcal{R}esTestPar_i$ et $t_1 \in \mathcal{R}esTestPar_j$.
   On veut montrer que $r_i = r_j$ et $P_i = P_j$.
   Comme $t_1\ \mathcal{RIT}\ t_2$, d'après le point 1 de la définition 5.2.1, $r_i = r_j$ et $P_i = P_j$.
   □
2. il reste maintenant à prouver que $t_2 \Rightarrow \overline{t_1}$.
   Comme $t_1\ \mathcal{RIT}\ t_2$, d'après le point 2 de la définition 5.2.1, on a $t_1 \Rightarrow \overline{t_2}$.
   En utilisant la contraposée, cela équivaut à dire que $\overline{\overline{t_2}} \Rightarrow \overline{t_1}$.
   Or $\overline{\overline{t_2}} = t_2$. On a donc $t_2 \Rightarrow \overline{t_1}$. □

Les relations d'incompatibilité induisent alors des relations d'incompatibilité sur l'ensemble des comportements des tâches.
Nous disons que $\mathcal{RIT}$ engendre $\mathcal{RI}$, ce que nous notons $\mathcal{RIT} \rhd \mathcal{RI}$.

Définition 5.2.2 Comportements incompatibles
Deux **comportements** $Comp_{il} \in Compeff_i$ et $Comp_{i'l'} \in Compeff_{i'}$, $i \neq i'$, $i, i' \in \{1 \dots n\}$ sont **incompatibles**, ce que nous notons $Comp_{il}\ \mathcal{RI}\ Comp_{i'l'}$ si et seulement si $\exists\ t \in \mathcal{R}esTestPar_i, t' \in \mathcal{R}esTestPar_{i'}$ tel que t apparait dans $Comp_{il}$, t' apparait dans $Comp_{i'l'}$ et $t\ \mathcal{RIT}\ t'$.

Remarque 5.2.1 La relation d'incompatibilité $\mathcal{RT}$ est une relation symétrique.

Preuve 5.2.2 $\mathcal{RI}$ est symétrique
Cette propriété découle de la symétrie de $\mathcal{RIT}$. □

Ainsi, il suffit qu'il existe deux tests de $\mathcal{R}esTestPar$ incompatibles pour conclure que les comportements qui les contiennent sont incompatibles deux à deux.
Dans la pratique, plusieurs tests peuvent appartenir à un même comportement. Il peut donc se produire que deux comportements soient déclarés incompatibles au titre de plusieurs couples de tests.
Ils seront donc générés plusieurs fois à partir de $\mathcal{RIT}$.

Pour illustrer nos propos, considérons les 2 tâches T_1 et T_2 données figure 5.2 sous leur forme arborescente.
Remarquons que $Comp_{13}\ \mathcal{RI}\ Comp_{22}$ car $(x \geq 10)_{T_1}\ \mathcal{RIT}\ (x < 5)_{T_2}$.
De même, $Comp_{13}\ \mathcal{RI}\ Comp_{22}$ car $(y \geq 8)_{T_1}\ \mathcal{RIT}\ (y < 4)_{T_2}$.

Dans la suite, nous allons donner un algorithme de génération de $\mathcal{RI}$ à partir de $\mathcal{RIT}$.
Pour une meilleure efficacité, nous devons éviter autant que possible les redondances dans la génération.

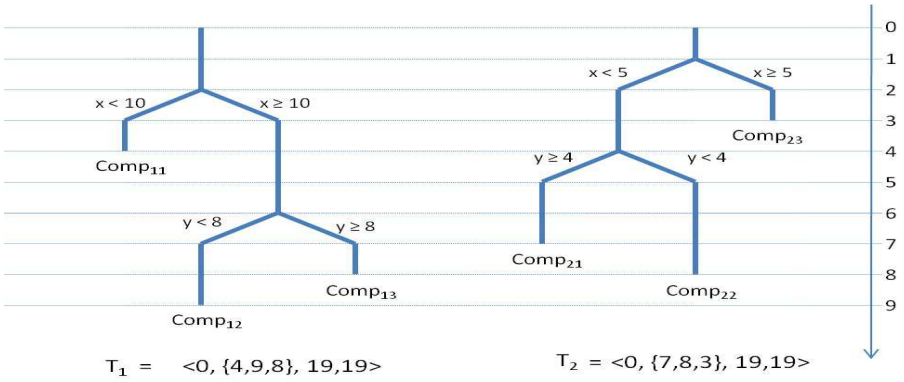


Figure 5.2 – Gestion efficace des comportements incompatibles

Si nous reprenons l'exemple de la figure 5.2, nous remarquons que $(Comp_{13}, Comp_{22})$ est générée deux fois : une fois au titre de $(x \geq 10)_{T_1}\ \mathcal{RIT}\ (x < 5)_{T_2}$ et une fois par $(y \geq 8)_{T_1}\ \mathcal{RIT}\ (y < 4)_{T_2}$.
En fait, la relation $(y \geq 8)_{T_1}\ \mathcal{RIT}\ (y < 4)_{T_2}$ n'apporte pas d'informations supplémentaires par rapport à la relation $(x \geq 10)_{T_1}\ \mathcal{RIT}\ (x < 5)_{T_2}$.
En effet, de $(x \geq 10)_{T_1}\ \mathcal{RIT}\ (x < 5)_{T_2}$, les relations suivantes sont établies :
$Comp_{12}\ \mathcal{RI}\ Comp_{21}$, $Comp_{12}\ \mathcal{RI}\ Comp_{22}$, $Comp_{13}\ \mathcal{RI}\ Comp_{21}$ et
$Comp_{13}\ \mathcal{RI}\ Comp_{22}$.
De même, de $(y \geq 8)_{T_1}\ \mathcal{RIT}\ (y < 4)_{T_2}$, on a $Comp_{13}\ \mathcal{RI}\ Comp_{22}$. Cette dernière relation était déjà connue.
Nous pouvons donc ignorer $(y \geq 8)_{T_1}\ \mathcal{RIT}\ (y < 4)_{T_2}$ pendant l'étude des relations incompatibles.

L'objectif de la section 5.2.2 est de définir une relation minimale $\mathcal{RIT}_{min}$ engendrant la même relation $\mathcal{RI}$.

5.2.2 Gestion des relations incompatibles

Dans la suite du document, nous voulons implémenter la relation $\mathcal{RIT}$, puis générer la relation $\mathcal{RI}$.
Il nous est donc apparu comme légitime, par souci d'efficacité, de chercher à minimiser $\mathcal{RIT}$.
De façon plus précise, nous cherchons un ensemble $\mathcal{RIT}_{min}$ minimal qui vérifie les propriétés données propriété 5.2.2.

Propriété 5.2.2 Caractéristiques de $\mathcal{RIT}_{min}$

1. $\mathcal{RIT}_{min} \subseteq \mathcal{RIT}$ ($\mathcal{RIT}_{min}$ est incluse ou égale à $\mathcal{RIT}$) ;
2. $\mathcal{RIT}_{min}$ induit la relation $\mathcal{RI}$ ($\mathcal{RIT}_{min} \rhd \mathcal{RI}$) ;
3. si on retire un élément de $\mathcal{RIT}_{min}$, la relation $\mathcal{RI}'$ engendrée est incluse strictement dans $\mathcal{RI}$ ($\mathcal{RI}' \subsetneq \mathcal{RI}$).

L'obtention de l'ensemble $\mathcal{RIT}$ relève de la partie mise en œuvre.
Nous supposons donc pour l'instant que cet ensemble est connu. Nous donnons au chapitre 8 une méthodologie d'obtention de $\mathcal{RIT}$.
Nous donnons méthode 5.2.1 une méthodologie de réduction de $\mathcal{RIT}$, afin d'obtenir $\mathcal{RIT}_{min}$.

Méthode 5.2.1 Principe de réduction de $\mathcal{RIT}$

1. nous partons de l'ensemble $\mathcal{RIT}$;
2. $\mathcal{RIT}_{min} := \emptyset$: la relation $\mathcal{RIT}_{min}$ est initialement vide ;
3. pour tout couple $(t^s_{il}, t^r_{jk}) \in \mathcal{RIT}$, soit Pre_i et Pre_j tels que $Pre_i.t^s_{il}$ et $Pre_j.t^r_{jk}$ soient les préfixes d'au moins un comportement de T_i et T_j respectivement. Pre_i et Pre_j sont les historiques. Ils correspondent à ce qu'ont fait les tâches avant d'atteindre respectivement t^s_{il} et t^r_{jk} :
   (a) si $\exists\ t^{s'}_{il'} \in \mathcal{R}esTestPar_i$ tel que $Pre_i = u.t^{s'}_{il'}.v$ et $(t^{s'}_{il'}, t^r_{jk}) \in \mathcal{RIT}$, alors $(t^s_{il}, t^r_{jk}) \notin \mathcal{RIT}_{min}$ et $\mathcal{RIT}_{min} := \mathcal{RIT}_{min} \cup \emptyset$;
   (b) la relation d'incompatibilité $\mathcal{RIT}$ étant symétrique (remarque 5.2.1), le cas symétrique, si $\exists\ t^{r'}_{jk'} \in \mathcal{R}esTestPar_j$ tels que $Pre_j = u'.t^{r'}_{jk'}.v'$, n'a plus à être traité ;
   (c) si $\exists\ t^{s'}_{il'} \in \mathcal{R}esTestPar_i$, et $\exists\ t^{r'}_{jk'} \in \mathcal{R}esTestPar_j$ tel que $Pre_i = u.t^{s'}_{il'}.v$ et $Pre_j = u'.t^{r'}_{jk'}.v'$, avec $u,\ v \in \alpha\beta_i$ et $u',\ v' \in \alpha\beta_j$, et tels que $(t^{s'}_{il'}, t^{r'}_{jk'}) \in \mathcal{RIT}$, alors $(t^s_{il}, t^r_{jk}) \notin \mathcal{RIT}_{min}$ et $\mathcal{RIT}_{min} := \mathcal{RIT}_{min} \cup \emptyset$;
   (d) sinon, $(t^s_{il}, t^r_{jk}) \in \mathcal{RIT}_{min}$: $\mathcal{RIT}_{min} := \mathcal{RIT}_{min} \cup \{(t^s_{il}, t^r_{jk})\}$.

Les points 3a, 3b et 3c permettent de ne pas ajouter à $\mathcal{RIT}_{min}$ certains éléments de $\mathcal{RIT}$.
Ces éléments vont en effet générer des comportements incompatibles, qui ont déjà été produits par des tests existants dans $\mathcal{RIT}_{min}$, situés en amont dans les tâches. C'est la raison pour laquelle l'ensemble $\mathcal{RIT}_{min}$ n'est pas modifié.
Le point 3d ajoute à $\mathcal{RIT}_{min}$ un couple de tests incompatibles, pour lequel il n'existe pas dans la liste de leur historique de tests incompatibles.
En cas de redondance, on privilégie ainsi les couples qui apparaissent les premiers.

En considérant l'exemple de la figure 5.2, nous avons :
$\mathcal{RIT} = \{((x \geq 10)_{T_1}, (x < 5)_{T_2}),\ ((y \geq 8)_{T_1},\ (y < 4)_{T_2})\}$.
L'ensemble $\mathcal{RI}$ induit par $\mathcal{RIT}$ est :
$\mathcal{RI} = \{(Comp_{12},\ Comp_{21}),\ (Comp_{12},\ Comp_{22}),\ (Comp_{13},\ Comp_{21}),$
$(Comp_{13},\ Comp_{22})\}$.
En utilisant méthode 5.2.1, l'ensemble $\mathcal{RIT}_{min}$ est :
$\mathcal{RIT}_{min} = \{((x \geq 10)_{T_1}, (x < 5)_{T_2})\}$.
Le couple $((y \geq 8)_{T_1},\ (y < 4)_{T_2}) \notin \mathcal{RIT}_{min}$.
Il correspond en effet au point 3c de la méthode 5.2.1 : le test $(x \geq 10)_{T_1} \in Pre_1$, où Pre_1 est un historique de $(y \geq 8)_{T_1}$.

De même, le test $(x < 5)_{T_2} \in Pre_2$, où Pre_2 est un historique de $(y < 4)_{T_2}$.
Enfin, nous avons $(x \geq 10)_{T_1}\ \mathcal{RIT}\ (x < 5)_{T_2}$. La méthode 5.2.1 préconise de ne pas ajouter $((y \geq 8)_{T_1},\ (y < 4)_{T_2})$ à $\mathcal{RIT}_{min}$.
L'ensemble $\mathcal{RI}$ induit par $\mathcal{RIT}_{min}$ est :
$\mathcal{RI} = \{(Comp_{12},\ Comp_{21}),\ (Comp_{12},\ Comp_{22}),\ (Comp_{13},\ Comp_{21}),$
$(Comp_{13},\ Comp_{22})\}$
$\mathcal{RIT}_{min}$ et $\mathcal{RIT}$ induisent le même ensemble $\mathcal{RI}$.

Comme nous l'avons déjà dit, l'objet des points 3a, 3b et 3c de la méthode 5.2.1 est d'éviter l'ajout à $\mathcal{RIT}_{min}$ de tests incompatibles qui induisent des comportements incompatibles déjà induits par certains tests de $\mathcal{RIT}_{min}$.

Il peut toutefois arriver que, malgré le fait que $\mathcal{RIT}_{min}$ soit bien construit, des redondances subsistent, c'est à dire qu'un même couple de comportements incompatibles soit généré plusieurs fois.
Ceci correspond aux cas où les relations d'incompatibilité comportementales induites par 2 couples distincts de $\mathcal{RIT}_{min}$ sont d'intersection non vide, mais chacune contient des couples qui lui sont propres.

L'exemple de la figure 5.3 illustre ce cas.

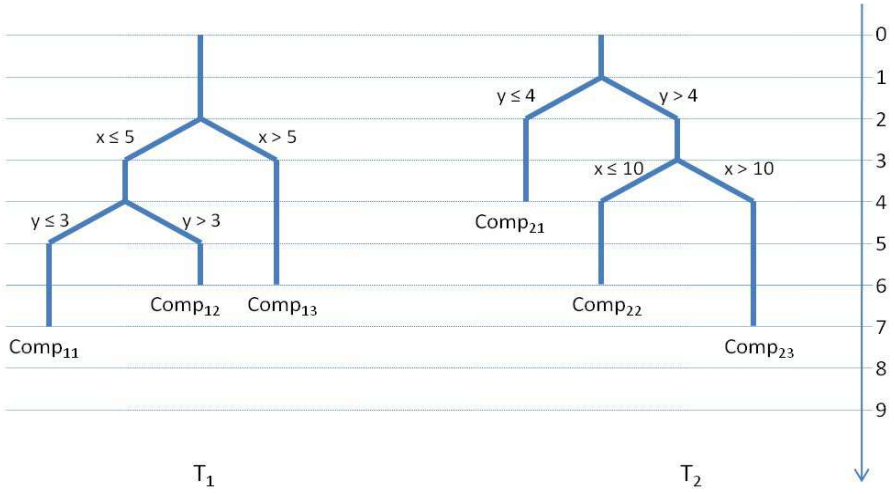


Figure 5.3 – Un cas où 2 comportements incompatibles identiques proviennent de 2 tests de $\mathcal{RIT}_{min}$

$\mathcal{RIT} = \{((x \leq 5)_{T_1}, (x > 10)_{T_2}),\ ((y \leq 3)_{T_1},\ (y > 4)_{T_2})\}$.
En appliquant le principe de réduction donné méthode 5.2.1, nous sommes, pour les deux couples de tests, dans le cas 3d.
Par conséquent,
$\mathcal{RIT}_{min} = \{((x \leq 5)_{T_1}, (x > 10)_{T_2}),\ ((y \leq 3)_{T_1},\ (y > 4)_{T_2})\}$.

Les couples de comportements incompatibles induits par $((x \leq 5)_{T_1}, (x > 10)_{T_2})$ sont : $(Comp_{11}, Comp_{23})$ et $(Comp_{12}, Comp_{23})$.
Les couples de comportements incompatibles induits par $((y \leq 3)_{T_1}, (y > 4)_{T_2})$ sont : $(Comp_{11}, Comp_{22})$ et $(Comp_{11}, Comp_{23})$.
Ces deux ensembles sont non vides, mais aucun n'est inclus dans l'autre.
Le couple $(Comp_{11}, Comp_{23})$ provient des deux éléments de $\mathcal{RIT}_{min}$.
Nous ne pouvons cependant pas éliminer ni $((x \leq 5)_{T_1}, (x > 10)_{T_2})$, ni $((y \leq 3)_{T_1}, (y > 4)_{T_2})$, car sinon des incompatibilités comportementales seraient perdues.

L'ensemble $\mathcal{RIT}_{min}$ étant construit, notre objectif est maintenant de vérifier qu'il est effectivement minimal au sens de la propriété 5.2.2.

Théorème 5.2.1 Minimalité de $\mathcal{RIT}_{min}$
L'ensemble $\mathcal{RIT}_{min}$ est minimal parmi les ensembles de résultats de tests incompatibles qui engendrent $\mathcal{RI}$.

Preuve 5.2.3 Minimalité de $\mathcal{RIT}_{min}$
Nous devons pour cela montrer que $\mathcal{RIT}_{min}$ vérifie bien les 3 points de la propriété 5.2.2 :

1. $\mathcal{RIT}_{min} \subseteq \mathcal{RIT}$.
   Par définition, $\mathcal{RIT}_{min}$ est construit en réduisant $\mathcal{RIT}$. Par conséquent, $\mathcal{RIT}_{min} \subseteq \mathcal{RIT}$. □
2. $\mathcal{RIT}_{min}$ induit la relation $\mathcal{RI}$.
   Soit $\mathcal{RI}_{min}$ la relation engendrée par $\mathcal{RIT}_{min}$.
   Nous allons prouver le lemme 5.2.1.

   Lemme 5.2.1 $\mathcal{RIT}_{min}$ engendre $\mathcal{RI}$: les ensembles $\mathcal{RI}_{min}$ et $\mathcal{RI}$ sont égaux.

   Preuve 5.2.4 $\mathcal{RIT}_{min}$ engendre $\mathcal{RI}$
   Nous définissons un opérateur de comparaison d'éléments de $\alpha\beta_i$.

   Définition 5.2.3 La relation préfixe $\propto$
   La **relation préfixe** $\propto$ compare 2 mots écrits sur un alphabet $\mathcal{A}$.
   Soient $u = \alpha_1.....\alpha_p$ et $v = \beta_1.....\beta_q$, avec $\alpha_1, \ldots, \alpha_p, \beta_1, \ldots, \beta_q \in \mathcal{A}$.
   Nous dirons que u est un préfixe de v, et nous noterons $u \propto v$ si :
   (a) $p \leq q$;
   (b) $\alpha_1 = \beta_1, \ldots, \alpha_p = \beta_p$.

   Pour prouver l'égalité entre $\mathcal{RI}_{min}$ et $\mathcal{RI}$, nous montrons les deux sens de l'inclusion.
   (a) Nous montrons d'abord que $\mathcal{RI}_{min} \subseteq \mathcal{RI}$, ce qui découle directement de $\mathcal{RIT}_{min} \subseteq \mathcal{RIT}$.
   Soit $(c, c') \in \mathcal{RI}_{min}$, alors $\exists\ (t, t') \in \mathcal{RIT}_{min}$ tel que $c = u.t.v$ et $c' = u'.t'.v'$, où u, v (u', v') sont des suites d'éléments de $\alpha\beta_i$ ($\alpha\beta_j$).
   Or, $\mathcal{RIT}_{min} \subseteq \mathcal{RIT}$, donc, $(t, t') \in \mathcal{RIT}$, et partant, $(c, c') \in \mathcal{RI}$, d'où $\mathcal{RI}_{min} \subseteq \mathcal{RI}$.

(b) Nous montrons ensuite que $\mathcal{RI} \subseteq \mathcal{RI}_{min}$.
Soit $(c, c') \in \mathcal{RI}$, alors $\exists\ (t, t') \in \mathcal{RIT}$ tel que $c = u.t.v$ et $c' = u'.t'.v'$.
Deux cas peuvent se produire :

i. soit $(t, t') \in \mathcal{RIT}_{min}$.
Dans ce cas, $(c, c') \in \mathcal{RI}_{min}$ par définition de $\mathcal{RI}_{min}$;

ii. soit $(t, t') \notin \mathcal{RIT}_{min}$.
Comme $(t, t') \notin \mathcal{RIT}_{min}$, par définition de $\mathcal{RIT}_{min}$,
$\exists\ (t_1, t'_1) \in \mathcal{RIT}$ tel que $c = u_1.t_1.v_1$ et $c' = u'_1.t'_1.v'_1$, avec
$u_1.t_1 \propto u.t$ et $u'_1.t'_1 \propto u'.t'$ et $|u_1.t_1| + |u'_1.t'_1| < |u.t| + |u'.t'|$ [1].
Si $(t_1, t'_1) \in \mathcal{RIT}_{min}$, $(c, c') \in \mathcal{RI}_{min}$.
Sinon, $\exists\ (t_2, t'_2) \in \mathcal{RIT}$ tel que $c = u_2.t_2.v_2$ et $c' = u'_2.t'_2.v'_2$, avec
$u_2.t_2 \propto u_1.t_1$ et $u'_2.t'_2 \propto u'_1.t'_1$ et $|u_2.t_2| + |u'_2.t'_2| < |u_1.t_1 + |u'_1.t'_1|$.
Si $(t_2, t'_2) \in \mathcal{RIT}_{min}$, $(c, c') \in \mathcal{RI}_{min}$.
Sinon, comme précédemment, $\exists\ (t_3, t'_3) \in \mathcal{RIT}$ tel que $c = u_3.t_3.v_3$ et $c' = u'_3.t'_3.v'_3$, avec
$u_3.t_3 \propto u_2.t_2$ et $u'_3.t'_3 \propto u'_2.t'_2$ et
$|u_3.t_3| + |u'_3.t'_3| < |u_2.t_2| + |u'_2.t'_2|$.
Nous construisons ainsi, de proche en proche, les suites u_n, t_n, v_n, u'_n, t'_n et v'_n telles que :
$c = u_n.t_n.v_n$ et $c' = u'_n.t'_n.v'_n$, avec
$u_n.t_n \propto u_{n-1}.t_{n-1}$ et $u'_n.t'_n \propto u'_{n-1}.t'_{n-1}$ et
$|u_n.t_n| + |u'_n.t'_n| < |u_{n-1}.t_{n-1}| + |u'_{n-1}.t'_{n-1}|$.
$|u_n.t_n| + |u'_n.t'_n|$ est une suite entière strictement décroissante (par construction, nous avons
$|u_n.t_n| + |u'_n.t'_n| < |u_{n-1}.t_{n-1}| + |u'_{n-1}.t'_{n-1}|$) et minorée (0 est un minorant de cette suite).
Elle est donc convergente vers sa borne inférieure
$Inf\{|u_n.t_n|+|u'_n.t'_n|\} = |u_{n_0}.t_{n_0}|+|u'_{n_0}.t'_{n_0}|$, et est donc stationnaire.
On a donc $c = u_{n_0}.t_{n_0}.v_{n_0}$ et $c' = u'_{n'_0}.t'_{n'_0}.v'_{n'_0}$.
Pour conclure que $(c, c') \in \mathcal{RI}_{min}$, il suffit de montrer que
$(t_{n_0}, t'_{n'_0}) \in \mathcal{RIT}_{min}$.
Nous procèdons par l'absurde.
Supposons que $(t_{n_0}, t'_{n'_0}) \notin \mathcal{RIT}_{min}$, alors
$\exists\ (t_{m_0}, t'_{m'_0}) \in \mathcal{RIT}$ tel que $c = u_{m_0}.t_{m_0}.v_{m_0}$ et
$c' = u'_{m'_0}.t'_{m'_0}.v'_{m'_0}$ avec $u_{m_0}.t_{m_0} \propto u_{n_0}.t_{n_0}$ et
$u'_{m'_0}.t'_{m'_0} \propto u'_{n'_0}.t'_{n'_0}$ et
$|u_{m_0}.t_{m_0}| + |u'_{m'_0}.t'_{m'_0}| < |u_{n_0}.t_{n_0}| + |u'_{n'_0}.t'_{n'_0}|$.
Ce qui contredit le fait que $|u_{n_0}.t_{n_0}| + |u'_{n'_0}.t'_{n'_0}|$ soit égal à
$Inf\{|u_n.t_n| + |u'_n.t'_n|\}$.
Nous concluons donc que $(t_{n_0}, t'_{n'_0}) \in \mathcal{RIT}_{min}$, et partant
$(c, c') \in \mathcal{RI}_{min}$.

[1] $|u|$ est la longueur du mot u.

Dans tous les cas, $(c, c') \in \mathcal{RI}_{min}$.
Par conséquent $\mathcal{RI} \subset \mathcal{RI}_{min}$.

Conclusion : $\mathcal{RI} = \mathcal{RI}_{min}$. $\square_{Lemme 5.2.1}$

3. $\mathcal{RIT}_{min}$ est minimal.
Pour le prouver, nous allons montrer que, si on considère un ensemble de résultats de tests incompatibles strictement inclus dans $\mathcal{RIT}_{min}$, alors, cet ensemble n'est pas suffisant pour obtenir $\mathcal{RI}$.
Plus formellement, soit $(t_1, t_2) \in \mathcal{RIT}_{min}$, soit $\mathcal{RIT}' = \mathcal{RIT}_{min} \setminus \{(t_1, t_2)\}$.
Il s'agit de montrer que la relation $\mathcal{RI}'$ engendrée par $\mathcal{RIT}'$ vérifie
$\mathcal{RI}' \subsetneq \mathcal{RI}$, où $\subsetneq$ désigne l'inclusion stricte.
Deux choses sont à montrer :

   (a) $\mathcal{RI}' \subseteq \mathcal{RI}$.
   Soit $(c, c') \in \mathcal{RI}'$.
   Nous voulons montrer que $(c, c') \in \mathcal{RI}$.
   Comme $(c, c') \in \mathcal{RI}'$, alors $\exists\ (t, t') \in \mathcal{RIT}'$ tel que $c = u.t.v$ et
   $c' = u'.t'.v'$.
   Or $\mathcal{RIT}' \subset \mathcal{RIT}_{min}$, donc $(t, t') \in \mathcal{RIT}_{min}$, et $\mathcal{RIT}_{min} \rhd \mathcal{RI}$ (voir preuve 5.2.4).
   Par conséquent, $(c, c') \in \mathcal{RI}$.

   (b) $\mathcal{RI}' \neq \mathcal{RI}$.
   Nous définissons au préalable un opérateur d'occurrence d'un élément d'un alphabet $\mathcal{A}$ dans un mot écrit sur $\mathcal{A}$.

   Définition 5.2.4 L'opérateur ϱ
   Nous disons qu'un élément $a \in \mathcal{A}$ **apparait dans un mot**
   $m = m_1.m_2.\ldots.m_p$ écrit sur $\mathcal{A}$, $m_i \in \mathcal{A}$, et nous notons $a\ \varrho\ m$, si et seulement si $\exists\ i_0 \in 1 \ldots p$ tel que $a = m_{i_0}$.

   L'opérateur $\overline{\varrho}$ est la négation de ϱ.

   Définition 5.2.5 L'opérateur $\overline{\varrho}$
   Nous disons qu'un élément $a \in \mathcal{A}$ **n'apparait pas dans un mot**
   $m = m_1.m_2.\ldots.m_p$ écrit sur $\mathcal{A}$, $m_i \in \mathcal{A}$, et nous notons $a\ \overline{\varrho}\ m$, si et seulement si $\forall\ i \in 1 \ldots p$, $a \neq m_i$.

   Notre objectif est de montrer qu'il existe au moins un élément de $\mathcal{RI}$ qui n'est pas dans $\mathcal{RI}'$.

   Comme $(t_1, t_2) \in \mathcal{RIT}_{min}$, $\exists\ (c_0, c'_0) \in \mathcal{RI}$ tel que $c_0 = u_0.t_1.v_0$ et
   $c'_0 = u'_0.t_2.v'_0$, où u_0, v_0 (u'_0, v'_0) sont des mots écrits sur l'alphabet $\alpha\beta_i$ ($\alpha\beta_j$).
   Soit $\mathcal{T}(c_0, c'_0) = \{(t, t')\ tel\ que\ t\ \varrho\ c_0,\ t'\ \varrho\ c'_0\ et\ (t, t') \in \mathcal{RIT}_{min}\}$
   Nous allons discuter suivant le cardinal de $\mathcal{T}(c_0, c'_0)$:

i. $|\mathcal{T}(c_0, c'_0)| = 0$.
Ce cas est impossible, car le couple (t_1, t_2) est élément de $\mathcal{T}$;

ii. $|\mathcal{T}(c_0, c'_0)| = 1$.
Dans ce cas, l'unique élément de $\mathcal{T}$ est (t_1, t_2).
Par conséquent, (c_0, c'_0) est un couple de comportements induit par l'unique couple de tests (t_1, t_2), qui n'appartient pas à $\mathcal{RIT}'$.
Donc $(c_0, c'_0) \notin \mathcal{RI}'$;

iii. $|\mathcal{T}(c_0, c'_0)| > 1$.
Il existe donc au moins un autre couple de tests $(t_0, t'_0) \in \mathcal{T}$, tel que $(t_0, t'_0) \neq (t_1, t_2)$:

A. on ne peut pas avoir $t_0 \varrho u_0.t_1$ et $t'_0 \varrho u'_0.t_2$, car cela contredirait le fait que $(t_1, t_2) \in \mathcal{RIT}_{min}$ (voir méthode 5.2.1) ;

B. on ne peut pas avoir $t_0 \varrho v_0$ et $t'_0 \varrho v'_0$, car, sinon on ne pourrait pas avoir $(t_0, t'_0) \in \mathcal{RIT}_{min}$;

C. on a donc
soit
$$cas\ 1\ :\ \begin{cases} t_0 \varrho u_0.t_1 \\ t'_0 \varrho v'_0 \end{cases}$$
soit
$$cas\ 2\ :\ \begin{cases} t_0 \varrho v_0 \\ t'_0 \varrho u'_0.t_2 \end{cases}$$
Les deux cas étant symétriques, nous choisissons de traiter le cas 1.
Supposons donc que $t_0 \varrho u_0.t_1$.
Alors, $\forall\ t$ tel que $(t_0, t) \in \mathcal{T}(c_0, c'_0)$, on a $t \varrho v'_0$.
Soit t_1^j le premier de ces t (c'est à dire celui qui apparait en premier dans v'_0), on a donc
$v'_0 = m'_1.t_1^j.m''_1$ et m'_1 ne contient aucun test t tel que
$(t_0, t) \in \mathcal{T}(c_0, c'_0)$.
Nous construisons un nouveau comportement $c'_1 = u'_0.t_2.m'_1.\overline{t_1^j}.v'_1$.

Lemme 5.2.2 c'_1 est un comportement (effectif) de T_j.

Preuve 5.2.5 Pour cela, nous allons montrer dans un premier temps que $\forall\ t \varrho u'_0.t_2.m'_1$, t et $\overline{t_1^j}$ ne sont pas incompatibles.
Nous procédons par l'absurde.
Si t et $\overline{t_1^j}$ sont incompatibles, alors $t \Rightarrow t_1^j$.
On déduit que
$t_2 \wedge t \Rightarrow t_1^j$.
D'autre part, t_2 et t_1^j sont incompatibles. Donc $t_2 \Rightarrow \overline{t_1^j}$.
On déduit que $t_2 \wedge t \Rightarrow \overline{t_1^j}$.
En combinant les deux implications déduites, on a

$t_2 \wedge t \Rightarrow t_1^j \wedge \overline{t_1^j} = False$, donc $t_2 \Rightarrow \bar{t}$, c'est à dire $(t_2, t) \in \mathcal{RIT}_j$.
Ce qui est impossible, car, t_2 et $t \ \varrho \ c'_0$, et c'_0 est un comportement effectif.

Ensuite, nous avons vu lors de la construction des comportements effectifs, que l'on pourra toujours choisir après $\overline{t_1^j}$, à chaque fois que l'on trouvera un test, une alternative compatible avec tout ce qui précède.
Nous pouvons donc compléter le comportement pour aboutir à c'_1 qui est un comportement effectif de la forme voulue.
$\square_{Lemme 5.2.2}$

Nous travaillons donc maintenant sur le nouveau couple de comportements

$$\begin{cases} c_1 = u_0.t_1.v_1 \ (v_1 = v_0) \\ c'_1 = u'_0.t_2.m'_1.\overline{t_1^j}.v'_1 \end{cases}$$

avec $\forall \ t \ \varrho \ u'_0.t_2.m'_1$, $(t_0, t) \notin \mathcal{T}(c_1, c'_1)$ et $(t_1, t_2) \in \mathcal{T}(c_1, c'_1)$.
Si $|\mathcal{T}(c_1, c'_1)| > 1$, nous itérons le procédé :
soit t_2^j le premier test tel que $t_2^j \ \varrho \ v'_1$
$\exists \ t_0^2 \ \varrho \ u_0.t_1$ tel que $(t_0^2, t_2^j) \in \mathcal{T}(c_1, c'_1)$.
Le cas dual se traite de la même façon.
Nous pouvons donc construire soit le couple de comportements

$$\begin{cases} c_2 = u_0.t_1.v_2 \ (v_2 = v_1, \ donc \ c_2 = c_1) \\ c'_2 = u'_0.t_2.m'_1.\overline{t_1^j}.m'_2.\overline{t_2^j}.v'_2 \end{cases}$$

soit le couple de comportement

$$\begin{cases} c_2 = u_0.t_1.m_1.\overline{t_2^i}.v_2 \\ c'_2 = c'_1 \end{cases}$$

Nous construisons ainsi une suite de tests $(t_p^{i_p})$, $i_p \in \{i, j\}$ et une suite de comportements (c_p, c'_p) tel que $t_p^{i_p} \neq t_q^{i_q}$ si $p \neq q$ et
$c_p \ \mathcal{RI} \ c'_p$.
À chaque étape :
soit $|\mathcal{T}(c_p, c'_p)| = 1$, dans ce cas $(c_p, c'_p) \in \mathcal{RI}$, mais $(c_p, c'_p) \notin \mathcal{RI}'$ car (t_1, t_2) est l'unique couple de tests incompatibles ;
soit nous construisons le test $t_{p+1}^{i_p}$ qui est distinct de tous les éléments précédents de la suite.
Comme le nombre d'éléments de l'ensemble $\mathcal{T}est$ est fini, la construction ne peut se poursuivre indéfiniment.
Donc, $\exists \ p_0 < |\mathcal{T}est_i| + |\mathcal{T}est_j|$ tel que $|\mathcal{T}(c_{p_0}, c'_{p_0})| = 1$ et partant $(c_{p_0}, c'_{p_0}) \in \mathcal{RI} \setminus \mathcal{RI}'$. $\square$

Dans tous les cas, nous avons trouvé un élément de $\mathcal{RI}$ qui n'est pas dans $\mathcal{RI}'$, donc nous avons $\mathcal{RI}' \neq \mathcal{RI}$.

$\mathcal{RIT}_{min}$ est donc minimal parmi les ensembles de résultats de tests incompatibles qui engendrent $\mathcal{RI}$. $\square_{Théorème 5.2.1}$

Remarque 5.2.2 Pour une application temps-réel donnée, $\mathcal{RIT}_{min}$ est unique. En effet, les étapes de la construction de l'ensemble $\mathcal{RIT}_{min}$ sont déterministes.

Ces résultats sur la gestion des relations incompatibles sur l'ensemble des résultats des tests étant établis (l'obtention de $\mathcal{RIT}_{min}$), la construction de $\mathcal{RI}$ est donnée méthode 5.2.2.

Méthode 5.2.2 Construction des comportements incompatibles
Pour chaque couple $(t,\ t')$ de $\mathcal{RIT}_{min}$:

1. nous récupérons l'ensemble des comportements $Comp(t)$ qui contiennent le test t pour la tâche concernée ;
2. nous récupérons l'ensemble des comportements $Comp(t')$ qui contiennent le test t' pour la tâche concernée ;
3. nous ajoutons à $\mathcal{RI}$ tous les éléments de l'ensemble $Comp(t)XComp(t')$ qui ne sont pas déjà dans $\mathcal{RI}$.
   La vérification de la non redondance reste nécessaire, car nous avons vu que toutes les redondances ne peuvent pas être supprimées.

5.3 Contraintes structurelles

Après les contraintes d'ordre sémantique que nous venons d'étudier (section 5.2), nous nous intéressons dans cette section aux contraintes d'ordre structurel.
Il s'agit ici d'affiner et de compléter les contraintes énoncées section 1.2.2.2 qui portent sur les aspects structurels de l'application : l'échange de messages et le partage de ressources.
L'objectif est d'assurer la cohérence structurelle de l'application, afin d'éviter les pertes de messages et les situations d'inter-blocage.
Nous complétons les règles à respecter par l'application :

1. un message émis à l'intérieur d'un comportement d'une tâche ne peut pas être reçu à l'intérieur d'un comportement incompatible d'une autre tâche.
   De plus, il doit être reçu par tous les comportements compatibles de la tâche réceptrice ;
2. si un message est émis à l'intérieur d'un comportement d'une tâche, il doit être émis par tous les comportements de la tâche ;
3. si une ressource est prise à l'intérieur d'un comportement d'une tâche, elle est libérée à l'intérieur de ce même comportement.

Par la suite, nous utiliserons la notion d'application structurellement valide.

Définition 5.3.1 Application structurellement valide
Nous dirons qu'une application temps-réel est **structurellement valide** si :

1. elle n'est constituée que de tâches correctes (voir définition 4.1.2) ;
2. elle vérifie les points 1, 2 et 3 de la section 5.3.

5.4 La tâche oisive

La **tâche oisive** est une tâche fictive qui permet de modéliser l'inactivité du processeur.
Lorsqu'elle est ordonnancée, cela signifie que le processeur reste inactif.
Nous nous intéressons à cette tâche dans un contexte où la charge processeur est strictement plus petite que le nombre de processeurs (dans notre cas, $U < 1$, puisque nous sommes en environnement monoprocesseur).
Notre objectif est d'éviter d'obtenir pendant l'analyse d'ordonnançabilité uniquement des ordonnancements conservatifs[2].
L'utilisation de la tâche oisive permet d'insérer des temps creux même s'il y a des tâches prêtes. Ceci peut par exemple permettre de parer à certaines situations de blocages ou d'inversion de priorité.
Pour mettre en évidence l'impact de l'existence d'instructions conditionnelles dans la définition de la tâche oisive, nous distinguons deux cas :

1. les tâches de l'application n'ont pas d'instructions conditionnelles ;
2. au moins une tâche de l'application a une instruction conditionnelle.

5.4.1 Les tâches n'ont pas d'instructions conditionnelles

Lorsque le taux d'utilisation du processeur U est inférieur à 1, le processeur reste inactif pendant $P(1-U)$ unités de temps à chaque hyperpériode, où P est l'hyperpériode.
Ces temps d'inactivité sont appelés **temps creux cycliques**.
Lorsque l'on considère les ordonnancements en-ligne classiques, les temps creux sont placés au plus tard, puisque ces ordonnancements sont **conservatifs** : il n'y a de temps creux que s'il n'y a aucune tâche prête à être exécutée.
Cependant, les stratégies conservatives ne sont pas optimales dès lors que l'on considère des tâches dépendantes.

L'exemple 5.4.1.1 tiré de [Grolleau 1999] l'illustre bien.

Exemple 5.4.1.1 Non optimalité des ordonnancements conservatifs
Soit le système $S = \{T_1, T_2\}$ de 2 tâches T_1 et T_2, qui utilisent la même ressource R.

[2] Nous rappelons qu'un algorithme d'ordonnancement est conservatif s'il ne laisse le processeur inactif que quand il n'y a pas de tâches prêtes à être exécutées

$T_1 = < 0, 2, 4, 4 >$ et la ressource R est utilisée pendant 2 unités de temps, soit à chaque fois que T_1 s'exécute.
$T_2 = < 0, 1, 1, 5 >$ et la ressource R est utilisée pendant 1 unité de temps, soit à chaque fois que T_2 s'exécute.
Le système S est ordonnançable : l'hyperpériode est 20, et l'ordonnancement donné figure 5.4 est valide.

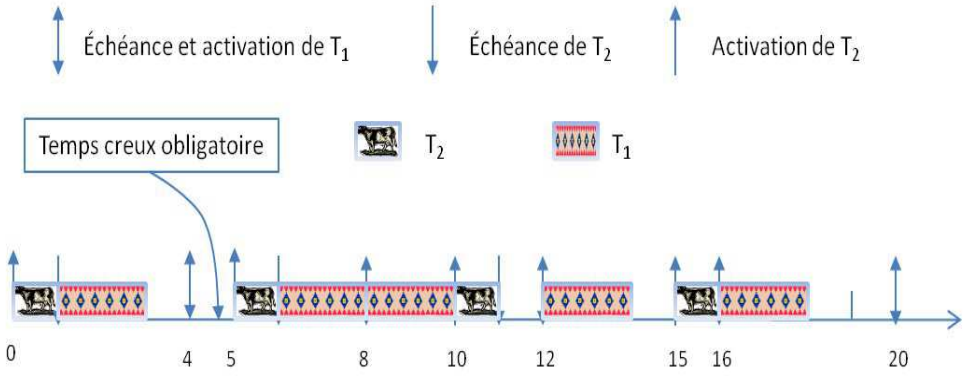


Figure 5.4 – Ordonnancement valide de S : les tâches respectent leur échéance. Par contre, le temps creux est obligatoire

L'ordonnancement de la figure 5.4 n'est pas conservatif.
En effet, la tâche T_1, bien que réveillée et prête ne peut être activée, car dans ce cas, la tâche T_2 ratera son échéance.
Une illustration en est faite figure 5.5.

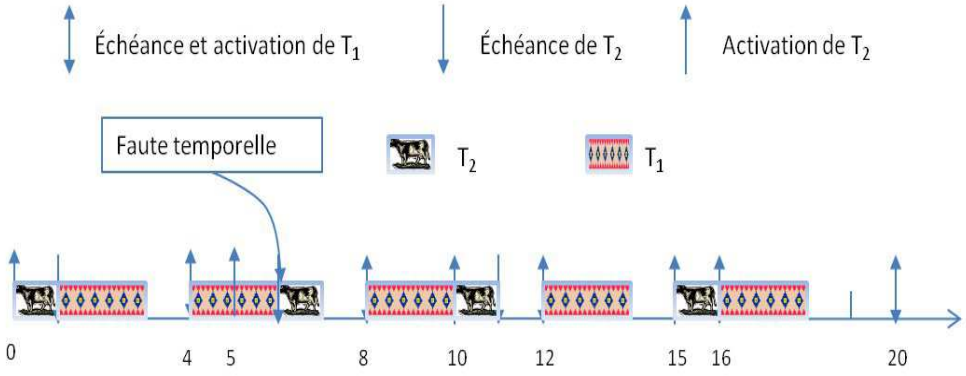


Figure 5.5 – Ordonnancement conservatif non valide de S : la tâche T_2 ne respecte pas son échéance

Plus généralement, nous ne pouvons pas trouver une stratégie conservative qui ordonnance cette application.
En effet, s'il n'y a pas de temps creux entre les instants 4 et 5, la tâche T_2 ratera

son échéance.
Nous pouvons donc conclure à la non optimalité des ordonnancements conservatifs dès que les tâches dépendantes sont considérées.

L'un des atouts des stratégies hors-ligne est de permettre la construction de séquences d'ordonnancement dans lesquelles les temps creux peuvent être placés où l'on veut.
Nous notons T_0, la **tâche oisive** dédiée à leur modélisation :

1. si les tâches sont à départs simultanés, les temps creux cycliques sont modélisés par la tâche $T_0 : < r_0 = 0, C_0 = P(1-U), D_0 = P_0 = P >$, et sont les seuls temps creux existants ;
2. dans le cas où elles sont à départs différés, on peut trouver en plus des temps creux cycliques, d'autres temps creux, en nombre fini, qui résultent des perturbations liées à la montée en charge progressive du système [Choquet-Geniet 2004] : ce sont les **temps creux acycliques**.
   Ils sont pris en compte dans une tâche non périodique, appelée **tâche creuse**, $T_c : < r_c = 0,\ C_c = n_c,\ D_c = t_c >$ où :
   (a) n_c est le nombre de temps creux acycliques. n_c est borné, et peut être déduit de l'étude de l'application ;
   (b) t_c est la date du dernier temps creux acyclique qui se produit avant l'instant $Max(r_i) + P$, $i \in 1 \ldots n$.
   Cette date est calculable, et indépendante de la stratégie d'ordonnancement choisie.
   Dans ce cas, la date de réveil de la tâche oisive T_0 est choisie de manière à ce que le système entre le plus vite possible dans son comportement cyclique : $r_0 = t_c + 1$.
   Les paramètres temporels de T_0 sont donc :
   $< r_0 = t_c + 1, C_0 = P(1-U), D_0 = P_0 = P >$.

5.4.2 Prise en compte des conditionnelles

En présence de tâches conditionnelles, le facteur d'utilisation U du système est variable : il dépend des comportements choisis par les tâches durant leur exécution. Par conséquent, la tâche oisive doit tenir compte de toutes ces fluctuations.
Nous adoptons l'**approche de la plus courte durée** issue de [Pailler 2006] : elle consiste à considérer la tâche oisive avec sa durée d'exécution minimale, qui correspond au facteur de charge maximal de l'application.
Nous choisissons donc dans chaque tâche conditionnelle le comportement qui a la plus longue durée d'exécution.
Si pendant son exécution, la tâche choisit un comportement qui ne correspond pas à la plus longue durée, afin d'atteindre le seuil (1) du facteur de charge, nous rallongeons la durée d'exécution de la tâche oisive du surplus.

La durée de la tâche oisive est donc réévaluée de façon dynamique au cours de l'application.
De cette façon, l'application ne risque jamais de se trouver dans un cas critique parce qu'elle a fait trop de temps creux[3].

Nous avons vu qu'en l'absence de conditionnelles, la tâche oisive était considérée lorsque le facteur de charge de l'application U était strictement plus petit que 1.
Avec l'approche de la plus courte durée en présence d'instructions conditionnelles, le facteur de charge U_{max}, correspondant au choix des plus longues branches dans chaque tâche, n'est pas suffisant pour conclure sur l'existence de la tâche oisive.
En effet, nous pouvons avoir une application pour laquelle $U_{max} \geq 1$, mais qui soit ordonnançable.
Cela peut provenir du fait que le U_{max} correspond à des choix de comportements incompatibles (voir chapitre 7 pour un exemple illustratif).
Dans ce cas, pendant son exécution, les comportements choisis par les tâches peuvent donner un facteur de charge U plus petit que 1.
Pour cette combinaison de comportement, une tâche oisive est nécessaire.
Nous distinguons donc deux cas :

1. si $P(1-U_{max}) \geq 0$, l'application possède une tâche oisive dont la durée d'exécution est initialement égale à $P(1-U_{max})$;
2. sinon, l'application peut posséder une tâche oisive.
   Nous étudions donc l'ensemble des combinaisons possibles de tous les comportements des tâches de l'application :
   (a) s'il en existe pour lesquels le facteur de charge U soit tel que $P(1-U) > 0$, alors, il faut prévoir une tâche oisive pour la modélisation complète de l'application.
   Le principe de construction de cette tâche est donné chapitre 8 ;
   (b) sinon, l'application ne comporte pas de tâche oisive.

   Plus simplement, nous prévoyons une tâche oisive dès que $U_{min} < 1$.

5.4.3 Prise en compte de la sémantique

La sémantique ici concerne les relations intra-tâches et inter-tâches.
La prise en compte explicite de ces relations peut influer sur le calcul du facteur de charge maximal et minimal.
Par conséquent, sur les paramètres à considérer pour la tâche oisive.
Il peut en effet arriver que, du fait des relations incompatibles, les comportements les plus longs des tâches ne puissent pas être considérés simultanément.
L'idéal serait dans ce cas de déterminer le U_{effmax} (le facteur de charge effectif

[3] Par opposition à la méthode de la plus longue durée où on attribue à la tâche oisive la durée maximale, qui correspond au facteur de charge minimal, et qui oblige à faire des retours arrières afin de ne pas dépasser le seuil 1 du facteur de charge de l'application, si une autre branche était choisie

maximal).
Ce travail s'avère très difficile, du fait de la gestion de l'historique des choix précédemment effectués.
C'est la raison pour laquelle, dans le cadre de cette thèse, nous nous limitons à la prise en compte des conditionnelles.
La prise en compte de la sémantique pour les tâches oisives fera l'objet d'une prochaine étude.

En appliquant l'approche retenue, le système de contrôle des essuie-glaces peut avoir une tâche oisive, car $P(1-U_{max}) = 11(1-\frac{11}{11}) = 0$, et le facteur de charge U correspondant par exemple aux comportements $Comp_{11}$, $Comp_{21}$ et $Comp_{31}$ est $U = \frac{10}{11}$, donc $P(1-U) = 11(1-\frac{10}{11}) = 1 > 0$.

5.5 Modélisation du système

Nous nous intéressons dans cette section à la modélisation complète de l'application.
Notre objectif est de proposer une méthodologie de modélisation complète d'une application temps-réel.
Nous voulons garantir que, partant du code source d'une application temps-réel, toutes les étapes sont bien considérées, afin d'obtenir le modèle *correct* de l'application qui intègre les aspects structurels, comportementaux et sémantiques.
Nous donnons méthode 5.5.1 les étapes à suivre pour obtenir le modèle final de l'application.

Méthode 5.5.1 Étapes de modélisation de l'application
Les étapes de modélisation d'une application sont :

1. modélisation de chaque tâche individuellement
   (a) extraire de chaque tâche l'ensemble des tests **Input** (qui ne sont pas modifiés par la tâche elle même) ;
   (b) calculer les éventuelles incompatibilités, afin de prendre en compte les relations intra-tâches ;
   (c) construire le modèle qui intègre les règles sémantiques de chaque tâche ;
2. modélisation de l'application complète
   (a) représenter les primitives temps-réel (échange de messages et partage des ressources) ;
   (b) calculer éventuellement la tâche oisive ;
   (c) calculer les éventuelles incompatibilités entre les différents tests des différentes tâches de l'application ;
   (d) construire le modèle final qui intègre les aspects sémantiques abordés dans cette thèse.

En appliquant la méthode 5.5.1 au système des essuie-glaces (exemple 1.2.2.1), les aspects suivants seront retenus pour la modélisation complète des essuie-glaces :

1. il n'y a pas de tests incompatibles à l'intérieur des tâches, donc pas de relations intra-tâches ;
2. le modèle complet de chaque tâche est donné chapitre 4 ;
3. l'ensemble des relations incompatibles est $\{((contact = 0)_{T_2}, (contact = 1)_{T_3}),\ ((contact = 1)_{T_2}, (contact = 0)_{T_3})\}$;
4. les comportements incompatibles qu'on obtient sont $\{(Comp_{21}, Comp_{32}),\ (Comp_{22}, Comp_{31})\}$;
5. il existe une tâche oisive à déterminer.
   Nous la déterminerons au chapitre 8.

Bilan

Dans ce chapitre, nous avons proposé une approche de modélisation des applications temps-réel.
Elle prend en compte de façon explicite les intructions conditionnelles, et permet de gérer les relations inter-tâches.
De plus, nous avons introduit la tâche oisive.
Elle nous permettra, pendant l'analyse d'ordonnançabilité, d'obtenir des ordonnancements non conservatifs.

Dans le chapitre 6, nous allons nous intéresser à la modélisation du comportement de l'application pendant son exécution.
Notre objectif sera l'obtention des ordonnancements réalistes, qui intègrent les instructions conditionnelles, et les relations inter-tâches.

Chapitre 6

Les arbres d'ordonnancement

"*Tout grand progrès scientifique est né d'une nouvelle audace de l'imagination*"
John Dewey

Sommaire

Lorsque l'objectif est la modélisation du comportement des applications pendant leur exécution, ce sont en général les séquences d'ordonnancement qui sont utilisées (voir chapitres 1 et 2).
Elles sont linéaires, et ne permettent pas de prendre explicitement en compte les instructions conditionnelles. Elles les encapsulent, en considérant leur comportement le plus long.
Certains auteurs ont proposé une approche arborescente qui pallie ce problème (voir chapitres 2 et 3).
La séquence d'ordonnancement est remplacée par un arbre d'ordonnancement ou par un ensemble de séquences d'ordonnancement, qui décrit le comportement ou l'ensemble des comportements explicites de l'application.
Ce chapitre a pour objectif de proposer des arbres d'ordonnancement sémantiquement corrects (qui gèrent les relations intra-tâches et inter-tâches).
Ainsi, les comportements des applications temps-réel pendant leur exécution seront modélisés de façon plus effective.
À notre connaissance, ce travail n'a jamais encore été fait.

Le travail présenté est réparti comme suit : nous commençons par décrire de façon formelle l'approche de représentation linéaire des séquences d'ordonnancement. Cela nous permet de mettre en place le vocabulaire que nous utiliserons pour définir les arbres d'ordonnancement.
Nous définissons ensuite de façon formelle les arbres d'ordonnancement, puis nous caractérisons les arbres valides : nous définissons les propriétés à respecter par un arbre, pour qu'il soit déclaré valide, suivant les aspects temporels, sémantiques et comportementaux.
Nous proposons enfin une première approche d'obtention des arbres ainsi caractérisés.

6.1 Ordonnancement linéaire

Dans un premier temps, nous revenons sur le cas linéaire, pour lequel nous définissons la notion de séquence d'ordonnancement.
Dans ce contexte, l'alphabet d'une tâche est
$\alpha\gamma_i = \{a_i\} \cup \{L(R), U(R) \textit{ tel que } R \in \mathcal{R}_i\} \cup \{S(M) \textit{ tel que } M \in \mathcal{EM}_i\} \cup \{R(M) \textit{ tel que } M \in \mathcal{RM}_i\}$, car il n'existe pas de tests conditionnels.
Une tâche s'écrit donc sous la forme $T_i = s_1.s_2.\ldots.s_k$, $s_l \in \alpha\gamma_i$ avec $l \in 1, \ldots, k$ et k est le nombre d'instructions de la tâche T_i.
La pire durée d'exécution de T_i est $C_i = \sum_{l=1}^{k} duree(s_l)$ où *duree* est la fonction définie section 4.2.3, et qui correspond au temps utilisé par la tâche pour exécuter les différentes actions.

En considérant l'exemple 1.2.2.1 des essuie-glaces, nous avons $T_1 = a_1.a_1.S(M_1)$ et $C_1 = 2$.

Nous allons maintenant caractériser le comportement des applications modélisées suivant leur représentation linéaire.
Pour cela, nous utilisons, comme cela se fait classiquement, les séquences d'ordonnancement.

6.1.1 Longueur des séquences d'ordonnancement

Les applications temps-réel considérées sont composées de tâches périodiques. Par conséquent, elles fonctionnent en régime permanent, puisque chaque tâche est activée périodiquement, et ce, potentiellement à l'infini.
Pour caractériser et valider les séquences d'ordonnancement, il suffit de les étudier sur un intervalle borné.
Cet intervalle est une fenêtre temporelle minimale sur laquelle la validation garantit l'absence définitive de faute temporelle.
Des études sur la cyclicité [Leung 1980] [Leung 1982] [Choquet-Geniet 2004] ont permis de calculer cette durée.
Les résultats suivants ont été établis.

Résultat 6.1.1 [Leung 1980] Tâches à départs simultanés
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 dans l'intervalle $[0, P]$, où P est l'hyperpériode.

Résultat 6.1.2 [Leung 1982] Tâches indépendantes à départs différés
Une configuration de n tâches indépendantes, à échéances inférieures ou égales aux périodes, est ordonnançable par **ED** si et seulement si la séquence d'ordonnancement produite est valide sur l'intervalle $[0,\ max_{i=1\ldots n}\{r_i\} + 2P]$.

Résultat 6.1.3 [Choquet-Geniet 2004] Tâches quelconques à départs quelconques
Un ordonnancement d'un système de tâches de charge $U \leq 1$ est valide si et seulement s'il est valide sur l'intervalle $[0,\ t_c + P + 1]$, où t_c est la date d'apparition du dernier temps creux acyclique.
Cette date dépend seulement de l'application, pas de la stratégie d'ordonnancement choisie et vérifie $t_c < max_{i=1\ldots n}\{r_i\} + P$.

Ces résultats nous conduisent à utiliser les notions suivantes pour caractériser les séquences d'ordonnancement.
Pour une application temps-réel $S = \{T_1, \ldots, T_n\}$ constituée de n tâches :

1. la séquence d'ordonnancement est cyclique, et de période $P = PPCM(P_i)$, $i \in 1 \ldots n$;
2. le début de cyclicité a lieu à l'instant $t_c + 1$, où t_c est la date du dernier temps creux acyclique.

6.1.2 Définition formelle des séquences d'ordonnancement

Soit $\alpha\gamma = \bigcup_{i=1}^{n} \alpha\gamma_i \cup \theta$, où θ représente un temps d'inactivité processeur, l'alphabet sur lequel nous allons décrire les séquences d'ordonnancement.
Nous étendons la fonction *duree* qui portait sur une tâche, par la fonction *Duree* sur l'application.
Elle nous permet de définir un morphisme entre l'ensemble des séquences d'ordonnancement et l'ensemble des entiers naturels.
Duree est définie par :

$$\begin{cases} Duree : \alpha\gamma \rightarrow \mathbb{N}\ telle\ que : \\ \theta \longmapsto 1 \\ a_i \longmapsto 1 \\ L(R), U(R), S(M), R(M) \longmapsto 0 \end{cases}$$

Notons que la fonction *Duree* traduit formellement ce qui a été dit jusqu'à présent :

1. un temps d'inactivité processeur correspond à une unité de temps du processeur à ne rien faire ;
2. les primitives temps-réel sont de durées supposées nulles ;
3. l'instruction élémentaire a_i est de durée 1.

Définition 6.1.1 Séquence d'ordonnancement
Une séquence d'ordonnancement $\mathcal{S}$ de n tâches $T_1, \ldots, T_n$, définie sur un intervalle $[0, T]$, est un mot $\mathcal{S} = s_1 \ldots . s_l$ écrit sur l'alphabet $\alpha\gamma$ tel que :

1. $\sum_{i=1}^{i=l} Duree(s_i) = T$;
2. pour tout $i \in 1 \ldots n$, $\forall\ p$ tel que $\sum_{j=1}^{p} Duree(s_j) \leq r_i$, nous avons $Proj_{\alpha\gamma_i}(s_1 \ldots . s_p) = \varepsilon$, où ε est le mot vide, $Proj_{\alpha\gamma_i}(s_1 \ldots . s_p)$ est la projection sur $\alpha\gamma_i$ de la séquence $s_1 \ldots . s_p$;
3. si k_1 et k_2 sont tels que
$$\begin{cases} \sum_{l=1}^{k_1-1} Duree(s_l) = r_i + (k-1) * P_i \\ \sum_{l=k_1}^{k_2} Duree(s_l) = D_i \end{cases}$$
alors $Proj_{\alpha\gamma_i}(s_{k_1} \ldots . s_{k_2}) = T_i$;
4. si k_1 et k_2 sont tels que
$$\begin{cases} \sum_{l=1}^{k_1-1} Duree(s_l) = r_i + (k-1) * P_i + D_i \\ \sum_{l=k_1}^{k_2} Duree(s_l) = r_i + k * P_i \end{cases}$$
alors $Proj_{\alpha\gamma_i}(s_{k_1} \ldots . s_{k_2}) = \varepsilon$.

Le point 1 traduit le fait qu'une séquence d'ordonnancement est définie sur l'intervalle complet $[0, T]$.
Le point 2 traduit le fait qu'une tâche ne s'exécute pas avant son activation.
Le point 3 traduit le fait qu'entre son activation et son échéance, une tâche est entièrement exécutée à l'intérieur d'une séquence d'ordonnancement.
Enfin, le point 4 traduit le fait qu'une tâche ne s'exécute pas entre son échéance et sa date de prochain réveil.
Notons enfin que la définition 6.1.1 caractérise les séquences d'ordonnancement temporellement valides, puisqu'elle intègre le respect des échéances.

Ensuite, nous devons vérifier que les séquences que nous considérons sont **structurellement valides**.
C'est à dire que, dans une séquence d'ordonnancement donnée :

1. tous les messages envoyés sont reçus ;
2. un message n'est pas reçu avant d'être envoyé ;
3. toutes les ressources prises sont libérées ;
4. une ressource n'est pas libérée avant qu'elle ne soit prise ;
5. les règles d'exclusion mutuelle sont vérifiées.

Dans la littérature on trouve plusieurs approches d'obtention des séquences d'ordonnancement [Xu 1992] [Zamorano 1997] [Grolleau 2002] [Largeteau 2002].
Ces approches ne seront pas détaillées dans ce travail.

Nous illustrons la notion de séquence d'ordonnancement sur l'exemple 6.1.2.1.

Exemple 6.1.2.1 Illustration de la notion de séquence d'ordonnancement
Nous considérons le système $S = \{T_1, T_2\}$ constitué de 2 tâches indépendantes T_1 et T_2 telles que :

1. $T_1 = < 0, 1, 2, 2 >$ et T_1 est décrite sur l'alphabet $\alpha\gamma_1$ par a_1 ;
2. $T_2 = < 0, 2, 4, 5 >$ et T_2 est décrite sur l'alphabet $\alpha\gamma_2$ par $a_2.a_2$.

La longueur d'étude (l'hyperpériode) est $PPCM(2, 5) = 10$.
Le facteur de charge $U = \frac{1}{2} + \frac{2}{5} = \frac{9}{10} \leq 1$.
Le système S peut être ordonnançable.
La séquence $\mathcal{S} = a_1.a_2.a_1.a_2.a_1.\theta.a_1.a_2.a_1.a_2$ est une séquence d'ordonnancement valide de S.
Elle est visualisée figure 6.1.

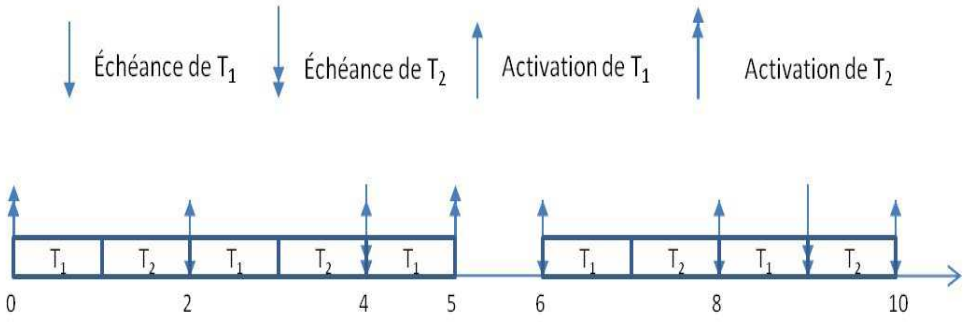


Figure 6.1 – Une séquence d'ordonnancement valide de S sur un processeur

6.2 Ordonnancement arborescent

Lorsque nous considérons des tâches avec instructions conditionnelles, le modèle linéaire de tâche est remplacé par un modèle arborescent, et de la même façon, le modèle linéaire d'exécution (la séquence d'ordonnancement) est remplacé par un modèle arborescent (l'arbre d'ordonnancement) que nous définissons ci-après.
Comme dans le cas des séquences d'ordonnancement, les arbres d'ordonnancement doivent être définis et validés sur une certaine longueur.
Dans le cas des arbres, nous parlerons de la profondeur.

6.2.1 Profondeur des arbres d'ordonnancement

Pour valider les arbres d'ordonnancement, [Pailler 2006] a généralisé les résultats obtenus dans le cas linéaire au cas arborescent.

Résultat 6.2.1 [Pailler 2006] Arbres d'ordonnancement
Soit un système de tâches ordonnançable. Quelque soit la stratégie d'ordonnancement 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 lorsqu'on considère la configuration correspondant aux comportements de durée maximale pour chaque instance de tâches.

Ainsi, nous retenons les notions suivantes pour nos arbres d'ordonnancement :

1. l'arbre d'ordonnancement est cyclique, de période $P = PPCM(P_i), i \in 1 \dots n$;
2. il rentre en régime permanent au pire à la date $t_c + 1$, avec t_c (date du dernier temps creux acyclique) calculée à partir de la séquence d'ordonnancement correspondant à la configuration C_{Max} (celle où on considère pour chaque tâche le comportement de durée maximale).

6.2.2 Définition formelle des arbres d'ordonnancement

Dans ce qui suit, les arbres de tâches considérés ne contiennent que les comportements effectifs, c'est à dire que les relations intra-tâches ont déjà été prises en compte.
Les arbres d'ordonnancement que nous construirons devront respecter les contraintes temporelles des tâches, et tenir compte des relations inter-tâches.
Avant de donner la définition formelle d'un arbre d'ordonnancement, notons que :

1. à un instant $t \geq r_i$, le numéro de l'instance en cours d'une tâche T_i est $\lfloor \frac{t-r_i}{P_i} \rfloor + 1$, où $\lfloor \frac{a}{b} \rfloor$ désigne la partie entière de $\frac{a}{b}$;
2. la clé d'un nœud correspond au temps écoulé depuis le lancement de l'application.

Ensuite, nous étendons la fonction *Duree*, en définissant une fonction $\mathcal{D}uree$ pour prendre en compte les tests conditionnels. Elle est donc définie sur l'alphabet $\alpha\beta$.
$\mathcal{D}uree$ est définie par :

$$\begin{cases} \mathcal{D}uree : \alpha\beta \rightarrow \mathbb{N} \textit{ telle que :} \\ \theta \longmapsto 1 \\ a_i \longmapsto 1 \\ L(R), U(R), S(M), R(M) \longmapsto 0 \\ t_{ij}^+, t_{ij}^- \longmapsto 1 \end{cases}$$

Notons que cette fonction est bien en accord avec notre hypothèse sur la durée des tests, qui est supposée égale à 1.

Nous définissons formellement les arbres d'ordonnancement.

Définition 6.2.1 Arbre d'ordonnancement
Un *arbre d'ordonnancement* $\mathcal{A}$ de n tâches $T_1 \dots T_n$, sur l'intervalle $[0, T]$, est un arbre dont les liens sont étiquetés par l'alphabet $\alpha\beta = \alpha\beta_1 \cup \dots \cup \alpha\beta_n \cup \theta$, où θ représente un temps d'inactivité du processeur, et les nœuds η contiennent des clés appartenant à $\mathbb{N}$.
Cet arbre vérifie les propriétés suivantes :

1. si η est la racine de l'arbre, alors $cle(\eta) = Temp(\eta) = 0$, où $Temp$ est la fonction définie section 4.2.3.
   Nous rappelons que $Temp(\eta)$ renvoie le temps mis depuis le début de l'application pour atteindre η ;
2. si $||fils(\eta)|| = 0$, alors η est un nœud terminal (une feuille), et $cle(\eta) = T$, où T est la durée d'étude.
   Il s'agit d'assurer que toutes les branches ont une profondeur égale à la période d'étude ;
3. si $||Fils(\eta)|| = 1$ alors
   (a) une tâche exécute une instruction qui n'est pas un test.
   Cela se traduit formellement par :
   soit $étiquette(lien(\eta, fils(\eta))) \in \{a_i, /i \in 1 \dots n\}$
   $\cup \{L(R), U(R)\ tel\ que\ R \in \mathcal{R}_i\}$
   $\cup \{S(M)\ tel\ que\ M \in \mathcal{EM}_i\} \cup \{R(M)\ tel\ que\ M \in \mathcal{RM}_i\} \cup \theta$
   et $cle(fils(\eta)) = cle(\eta) + \mathcal{D}uree(étiquette(lien(\eta, fils(\eta))))$;
   (b) une tâche exécute un test dont le résultat est corrélé à un résultat de test antérieur.
   Cela se traduit formellement par :
   soit $\exists\ t_{ij}$ et $s, s' \in \{+, -\}$ tel que $étiquette(lien(\eta, fils(\eta))) = t_{ij}^{s}$ et
   $cle(fils(\eta)) = cle(\eta) + 1$
   et $\exists\ \eta' \in ancetre(\eta)$ tel que :
      i. étiquette$(lien(Pere(\eta'), \eta')) = t_{kl}^{s'}$;
      ii. $t_{kl}^{s'}$ et $\overline{t_{ij}^{s}}$ sont incompatibles ;
      iii. $\lfloor \frac{cle(pere(\eta)) - r_i}{P_i} \rfloor + 1 = \lfloor \frac{cle(pere(\eta')) - r_k}{P_k} \rfloor + 1$.
      Cette égalité assure que la comparaison des tests se fait sur les mêmes numéros d'instances contenant les tests.
      Les tests portent donc sur les mêmes valeurs.
      Nous rappelons que si $t_{kl}^{s'}$ et $\overline{t_{ij}^{s}}$ sont incompatibles, alors $r_i = r_k$ et $P_i = P_k$;
4. une tâche exécute un test pour lequel les deux alternatives sont envisagées.
   Cela se traduit par :
   si $||fils(\eta)|| = 2$, alors $\exists\ t_{ij}^{+}$ et $t_{ij}^{-} \in \mathcal{T}_i$ tel que
   (a) étiquette$(lien(\eta, fils_1(\eta))) = t_{ij}^{+}$;
   (b) étiquette$(lien(\eta, fils_2(\eta))) = t_{ij}^{-}$.
   Dans ce cas, $cle(fils_i(\eta)) = cle(\eta) + 1$, $i \in \{1, 2\}$.

Le point 2 assure que l'arbre d'ordonnancement est construit sur sa durée de simulation.
Les points 3 et 4 assurent la prise en compte explicite des conditionnelles :

1. soit les deux comportements sont possibles (point 4) ;

2. soit un seul est possible (point 3b) car l'autre comportement est invalidé par un choix préalable réalisé dans une autre tâche, ou dans la même tâche, s'il s'agit d'une incompatibilité intra-tâche (dans ce cas $i = k$).

Le point 3a permet de prendre en compte les instructions non conditionnelles, à savoir les primitives temps-réel et les blocs indépendants.

Notons pour conclure cette définition sur les arbres d'ordonnancement, qu'ils caractérisent l'ensemble des comportements d'une application.
Un chemin de l'arbre correspond à un comportement de l'application.

Les arbres d'ordonnancement étant définis, nous voulons par la suite analyser le comportement de l'application à partir de ces arbres.
Pour cela, il est nécessaire de n'en retenir que les valides.
Nous nous intéressons section 6.3 à la caractérisation des arbres d'ordonnancement valides.

6.3 Caractérisation des arbres valides

Nous abordons quatre aspects de la validation des arbres : structurel, comportemental, sémantique et temporel.
Ces aspects permettent de caractériser complètement les applications temps-réel et constituent une synthèse de tous les aspects qui nous intéressent dans le cadre de cette thèse.

6.3.1 Validation structurelle

Nous voulons garantir ici que les arbres d'ordonnancement sont structurellement valides.
L'aspect structurel concerne le bon agencement des primitives temps-réel.
Il n'est pas pris en compte dans la définition 6.2.1 des arbres d'ordonnancement.
Nous supposons que les tâches sont structurellement valides, c'est à dire que les hypothèses données section 1.2.2.2 (sur la cohérence structurelle de l'application) et section 5.3 (sur les contraintes structurelles) sont respectées. Les applications considérées sont structurellement valides (définition 5.3.1).
Enfin, les tâches considérées sont sémantiquement correctes (définition 4.3.1).
Notre objectif est de garantir que pour chaque comportement de l'arbre d'ordonnancement retenu :

1. tous les messages émis sont effectivement reçus ;
2. il n'y a pas de réception d'un message avant qu'il ne soit envoyé ;
3. les ressources sont bien en exclusion mutuelle, c'est à dire que l'accès à une ressource se fait de façon exclusive.

Pour gérer cet aspect de la validation, nous associons, pour chaque comportement de l'application, un tableau avec un compteur, par message et par ressource présents

au sein de ce comportement.
Dans le cas des messages, le compteur est initialement nul, et dans le cas des ressources, il est initialement égal au nombre d'instances de la ressource.
Le compteur sera incrémenté de 1 dans le cas d'une libération de ressource ou d'un envoi de message, et décrémenté de 1 dans le cas d'une prise de ressource ou d'une réception de message.
Un **comportement** sera dit **structurellement valide**, si en fin du parcours, chaque compteur pour chaque message est nul, et chaque compteur de ressource est égal au nombre d'instances pour chaque ressource.
De plus, à chaque instant, les compteurs doivent toujours être supérieur ou égal à zero.
Remarquons que nous gérons de cette façon les ressources multi-instances.
Un **arbre** sera dit **structurellement valide** si tous ses comportements sont structurellement valides.

Soit $\mathcal{A}$ un arbre d'ordonnancement, et $\mathcal{C}omp$ l'ensemble de ses comportements.
Notre objectif est de proposer une méthodologie de vérification de la validité structurelle d'un arbre.
Soit $C \in \mathcal{C}omp$ un comportement de $\mathcal{A}$.
La méthode 6.3.1 permet de conclure de la validité structurelle de C.

Méthode 6.3.1 Validation structurelle
Nous supposons que les tâches sont sémantiquement correctes et que deux tâches qui communiquent ont la même date de réveil et la même période.
De plus, la fréquence d'émission de messages est égale à la fréquence de réception de messages entre ces deux tâches.
Le principe de validation est le suivant :

1. nous initialisons un compteur par ressource R : $V_0(R) = Nbre$, où $Nbre$ est le nombre d'instances de la ressource R.
   Pour une exclusion mutuelle simple, $Nbre = 1$;
2. nous initialisons un compteur par message M : $V_0(M) = 0$;
3. nous parcourons l'ensemble des instructions de C ;
   (a) si l'instruction considérée porte sur une ressource
      i. s'il s'agit d'une prise de ressource ($L(R)$), alors
         $V_0(R) := V_0(R) - 1$: le nombre d'instances disponibles de la ressource R est décrémenté.
         Si $V_0(R) < 0$, alors l'exclusion mutuelle n'est pas garantie.
         Le comportement n'est pas valide ;
      ii. s'il s'agit d'une libération de ressource ($U(R)$), alors
         $V_0(R) := V_0(R) + 1$: le nombre d'instances disponibles de la ressource R est incrémenté.
         Si $V_0(R) > Nbre$, alors la cohérence structurelle n'est pas garantie.
         Cela veut dire qu'une instance de la ressource R a été libérée avant

d'être prise.
Le comportement n'est pas valide ;

(b) si l'instruction considérée porte sur un message :

i. s'il s'agit d'un envoi de message ($S(M)$), alors $V_0(M) := V_0(M) + 1$;

ii. s'il s'agit d'une réception de message ($R(M)$), alors
$V_0(M) := V_0(M) - 1$.
Si $V_0(M) < 0$, alors la cohérence structurelle n'est pas garantie.
Cela veut dire que le message M a été reçu avant d'être envoyé.
Le comportement n'est pas valide ;

(c) sinon, nous passons à l'instruction suivante ;

4. si en fin de parcours du comportement, $V_0(R) = Nbre$ et $V_0(M) = 0$ pour toutes les ressources R et tous les messages M, alors le comportement C est structurellement valide ;

5. sinon, le comportement C n'est pas structurellement valide.

6.3.2 Validation comportementale et sémantique

Notre objectif est d'assurer que tous les comportements de l'application sont effectivement pris en compte dans les arbres d'ordonnancement.
Cela se traduit par la prise en compte des conditionnelles contenues dans les tâches, dans la définition des arbres d'ordonnancement.
Nous pouvons remarquer que cet aspect de la validation est traité au point 4 de la définition 6.2.1. Les instructions conditionnelles des tâches sont bien considérées.
Un arbre d'ordonnancement va être valide du point de vue comportemental, s'il intègre tous les comportements possibles de l'application.
La validation comportementale se fait pendant la construction des arbres (voir section 6.4).
La méthode de construction des arbres d'ordonnancement donnée section 6.4 préconise en effet qu'en présence d'un test conditionnel, trouvé dans une tâche, deux comportements sont alors à créer dans l'arbre.
Certains comportements peuvent ne pas apparaitre du fait des relations incompatibles (aspects sémantiques).
Cet aspect concerne le respect des relations inter-tâches dans les arbres d'ordonnancement.
Notre soucis est de retenir parmi tous les comportements possibles de l'arbre d'ordonnancement, uniquement ceux qui ne possèdent pas de tests incompatibles, instance par instance (voir définition 5.2.1).
L'arbre ainsi épuré de comportements incohérents sera dit sémantiquement valide.
Cet aspect de la validation est abordé définition 6.2.1 point 3b, et les arbres sémantiquement valides s'obtiennent par construction (voir section 6.4).
Enfin, nous nous intéressons à la validation temporelle.

6.3.3 Validation temporelle

Il s'agit ici de garantir que les contraintes temporelles des tâches constituant l'application sont respectées, c'est à dire que les tâches respectent leur échéance. Cet aspect n'est pas pris en compte dans la définition 6.2.1 des arbres d'ordonnancement. Un arbre d'ordonnancement $\mathcal{A}$ sera dit **temporellement valide** s'il respecte les points 1 et 2 suivants :

1. $\forall\, lien(pere(\eta),\eta)$, si $r_i+k*P_i+D_i < cle(\eta) \leq r_i+(k+1)*P_i$ ou si $cle(\eta) \leq r_i$ alors $étiquette(lien(pere(\eta),\eta)) \notin \alpha\beta_i$. Cela signifie qu'entre une échéance et l'activation suivante, la tâche ne s'exécute pas ;
2. la projection sur $\alpha\beta_i$ d'un chemin $\eta \rightarrow \eta'$ où η est un nœud qui vérifie :

$$\begin{cases} cle(\eta) = r_i + k * P_i \\ cle(pere(\eta)) < cle(\eta) \end{cases}$$

et η' est un nœud qui vérifie
soit :

$$\begin{cases} cle(\eta') = r_i + k * P_i + D_i \\ \eta' \ est\ une\ feuille \end{cases}$$

soit :

$$\begin{cases} cle(\eta') = r_i + k * P_i + D_i \\ cle(\eta') < cle(fils_1(\eta')) \end{cases}$$

correspond à un comportement valide de la tâche T_i.
Cela signifie qu'entre l'instant d'activation d'une instance et l'échéance suivante, la tâche exécute exactement l'un de ses comportements. Les conditions $cle(pere(\eta)) < cle(\eta)$ et $cle(\eta') < cle(fils_1(\eta'))$ permettent de garantir que l'on n'oublie pas de primitives temps-réel qui seraient en début ou en fin de tâches.

Les points 1 et 2 traduisent de façon générale le respect des dates d'activation et des échéances des tâches. Le point 1 exprime le fait qu'une tâche ne peut jamais être exécutée entre l'une de ses échéances et l'activation suivante. Le point 2 dit qu'entre l'activation d'une instance de la tâche et son échéance, la tâche a exécuté exactement l'un de ses comportements valides.

Dans toute la suite, quand nous dirons qu'un arbre est valide, cela voudra dire qu'il est valide sur les plans structurel, comportemental, sémantique et temporel.

Nous donnons figures 6.2 et 6.3 deux représentations de l'arbre
$\mathcal{A} = (a_1)^2.S(M_1).R(M_1).(a_2)^2$
$($
$t^+_{21}.a_2.S(M_2).R(M_2).a_3.t^+_{31}.(a_3)^2,$
$t^-_{21}.(a_2)^3.S(M_2).R(M_2).a_3.t^-_{31}.a_3$
$)$

où $(a_i)^n$ désigne la concaténation de a_i n fois.
Cet arbre est un arbre d'ordonnancement valide du système de contrôle-commande des essuie-glaces de l'exemple 1.2.2.1.

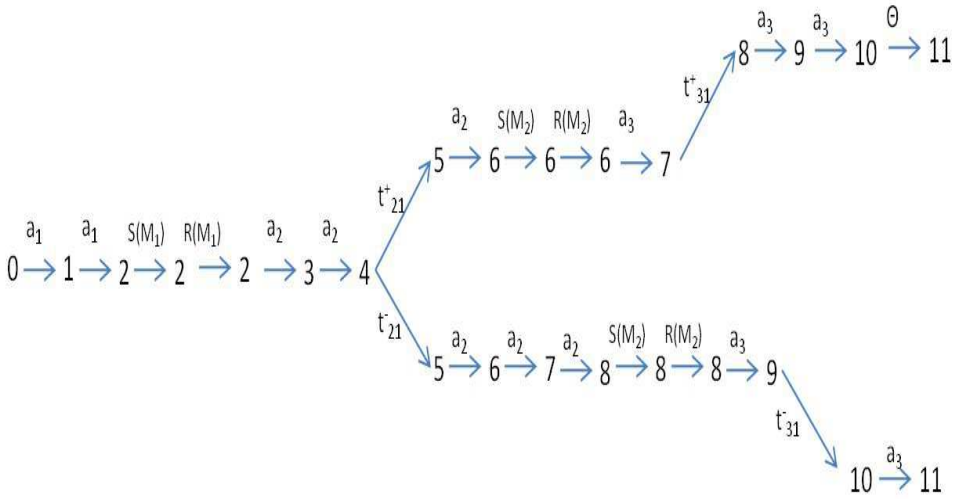


Figure 6.2 – Un arbre d'ordonnancement valide des essuie-glaces : θ représente un temps d'inactivité processeur

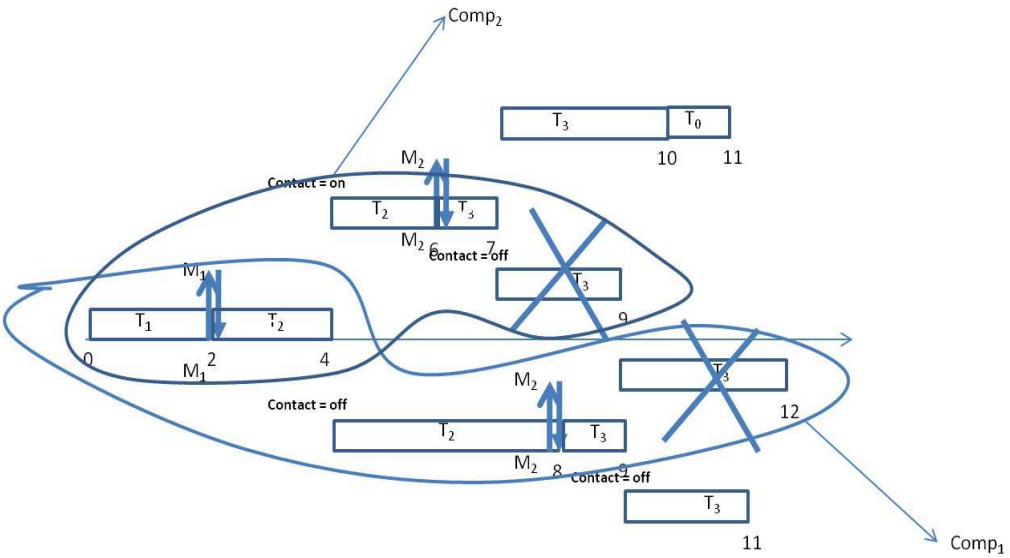


Figure 6.3 – Un arbre d'ordonnancement valide des essuie-glaces : T_0 est la tâche oisive

Cet arbre est construit et validé sur une profondeur $P = 11$ qui est l'hyperpériode, les tâches étant toutes à départs simultanés.

Les choix possibles représentés sur les tâches T_2 et T_3 montrent bien la prise en compte des aspects comportementaux.
Nous pouvons remarquer que le comportement $Comp_1$ qui aurait conduit à une invalidité temporelle (la charge processeur **U** plus grande que 1) est un cas impossible, car cela reviendrait à considérer simultanément la variable *contact* comme étant à *on* et à off.
De même, remarquons que l'analyse sémantique nous conduit à éliminer le comportement $Comp_2$.

Les arbres d'ordonnancement permettent donc de caractériser le comportement des applications temps-réel.
Nous nous intéressons section 6.4 à une méthodologie d'obtention de ces arbres.

6.4 Obtention des arbres d'ordonnancement

Nous proposons dans cette section un algorithme d'obtention des arbres d'ordonnancement que nous venons de caractériser.
Cet algorithme complète celui proposé dans [Pailler 2006] pour prendre en compte les aspects sémantiques (relations intra-tâches et inter-tâches).

6.4.1 Principe de l'algorithme

L'algorithme prend en entrée la liste des instances des tâches activées entre les instants 0 et T, qui correspond à la durée de simulation sur laquelle on veut construire les arbres d'ordonnancement.
Cette liste est obtenue par calcul direct sur les tâches données en entrée.
Les tâches sont données sous leur forme formelle (chaque tâche est définie sur l'alphabet $\alpha\beta_i$), et la liste des instances est initialement rangée en ordre croissant de date d'activation.
L'algorithme fournit en sortie l'ensemble des arbres d'ordonnancement sur l'intervalle [0, T].
La méthode de construction est conservative (c'est à dire qu'à chaque instant, l'algorithme ordonnance les tâches réveillées et prêtes à être exécutées) et incrémentale.
Le principe de l'algorithme est donné méthode 6.4.1.

Méthode 6.4.1 Construction des arbres d'ordonnancement
À l'étape courante, on dispose :

1. d'un ensemble de tâches ;
2. d'un ensemble de résultats de tests noté Mem.

Les étapes suivantes sont appliquées :

1. l'ensemble des tâches est partitionné en fonction de la première action b_i qui est issue des instances prêtes ;
2. pour chaque choix d'une action b_i :

(a) si b_i n'est pas un test :

i. nous construisons l'ensemble de tous les arbres obtenus avec le reste des actions à exécuter ;

ii. nous préfixons ces arbres par l'action b_i ;

(b) si b_i est un test :

i. si b_i^+ est incompatible avec un élément de Mem.
Nous construisons l'ensemble des arbres, en choisissant uniquement les comportements induits par b_i^-, et en mémorisant le fait qu'on a déjà fait un test, donc en refusant par la suite des comportements qui seraient incompatibles avec ce test (b_i^-) ;
Nous ajoutons donc b_i^- à Mem ;

ii. si b_i^- est incompatible avec un élément de Mem.
Nous construisons l'ensemble des arbres, en choisissant uniquement les comportements induits par b_i^+, et en mémorisant le fait qu'on a déjà fait un test, donc en refusant par la suite des comportements qui seraient incompatibles avec ce test (b_i^+) ;
Nous ajoutons donc b_i^+ à Mem ;

iii. si ni b_i^+, ni b_i^- n'est incompatible avec un élément de Mem :

A. nous construisons l'ensemble des arbres gauches, en choisissant les comportements induits par b_i^+, et en mémorisant le fait qu'on a déjà fait un test, donc en refusant par la suite des comportements qui seraient incompatibles avec ce test (b_i^+) ;

B. nous construisons l'ensemble des arbres droits, en choisissant les comportements induits par b_i^-, et en mémorisant le fait qu'on a déjà fait un test, donc en refusant par la suite des comportements qui seraient incompatibles avec ce test (b_i^-) ;

C. nous construisons tous les arbres en regroupant un arbre gauche et un arbre droit relié à la racine par les liens d'étiquettes respectives b_i^+ et b_i^- ;

3. l'ensemble ainsi obtenu est l'ensemble des arbres d'ordonnancement.

Ces arbres sont valides sur les plans sémantique et comportemental : le point 2b en assure la validité.
Il faut ensuite vérifier leur validité structurelle et temporelle (en utilisant les règles données sections 6.3.1 et 6.3.3) , afin de n'en retenir que les valides.
Nous formalisons la méthode 6.4.1 pour définir un générateur d'arbres d'ordonnancement.

6.4.2 Le générateur d'arbres

À un instant t, nous notons :

1. $T^{d_1}_{i_1,n_1}, \ldots T^{d_m}_{i_m,n_m}$ la liste des instances de tâches réveillées avant t, où i_j est le numéro de la tâche, n_i est le numéro de l'instance concernée, d_j est la date d'activation de $T^{d_j}_{i_j,n_j}$, $d_j = r_{i_j} + (n_j - 1) * P_{i_j}$.
   Pendant la construction des arbres, la liste des instances passée au générateur sera actualisée pas à pas ;
2. $e_i = \lceil \frac{t-r_i}{P_i} \rceil$ correspond au nombre d'instances de T_i activées dans l'intervalle $[0, t]$ et $m = \sum_i^n e_i$ est le nombre total d'instances activées avant t pour toutes les tâches ;
3. $Test$ est l'ensemble des tests conditionnels rencontrés pendant la construction de l'arbre. Il est initialement vide.
   Un test de $Test$ est de la forme $t^{n_{i_j},s}_{i_j,k}$ avec k le numéro du test, i_j le numéro de la tâche mère, n_{i_j} le numéro de l'instance contenant le test et s le signe du test.

Nous supposons que $d_1 \leq d_2 \leq \cdots \leq d_m$.
Si tel n'est pas le cas, il est toujours possible de réorganiser les instances afin d'avoir cet ordre.
Nous donnons définition 6.4.1 un opérateur de génération d'arbres d'ordonnancement.

Définition 6.4.1 Opérateur de génération des arbres
L'opérateur $\bigotimes_t$:{Ensemble d'instances réveillées avant t} X $Test \rightarrow$ {Ensemble des arbres d'ordonnancement de profondeur t} où t représente le temps restant pour atteindre la fin de la simulation, est défini par :

1. si $t = 0$, nous sommes en fin de construction des arbres.
   Deux cas peuvent se produire :
   (a) soit aucune instance réveillée ne commence par une primitive temps-réel, et dans ce cas nous ne construisons plus d'arbres.
   Nous formalisons le fait qu'on ne construit plus d'arbre par l'arbre vide ε :
   $\bigotimes_0([T^{d_j}_{i_j,n_j}]_{1 \leq j \leq m},\ Test) = \varepsilon$;
   (b) soit il existe parmi les instances réveillées certaines qui commencent par des primitives temps-réel.
   Nous savons donc construire d'autres arbres qui en tiennent compte sans qu'il n'y ait violation de contraintes temporelles, car les primitives temps-réel sont de durées supposées nulles.
   Si $\exists\, j > 0$ tel que $d_1 = d_2 = \ldots = d_j = 0$, et $T_{i_j} = a_{i_j}.T'_{i_j}$ où T'_{i_j} désigne le mot restant qui complète T_{i_j} et $a_{i_j} \in \{L(R), U(R), S(M), R(M)\}$, alors
   $\bigotimes_0([T^{d_j}_{i_j,n_j}]_{1 \leq j \leq m},\ Test) = \bigcup_{l=1}^{j} \{a_{i_l}.\lambda,\ tel\ que$
   $\lambda \in \bigotimes_0([T'^0_{i_l,n_l}, (T^0_{i_1,n_1}, \ldots, T^0_{i_j,n_j}) \backslash (T^0_{i_l,n_l}), T^{d_{j+1}}_{i_{j+1},n_{j+1}}, \ldots, T^{d_m}_{i_m,n_m}],\ Test)\}$.
   Ceci se traduit par :
      i. nous cherchons s'il existe des instances réveillées et qui commencent par des primitives temps-réel.
      S'il en existe, nous construisons un nouvel ensemble d'arbres ;

ii. chaque arbre commence par une primitive temps-réel, et est ensuite complété.
Le nouvel arbre λ est obtenu en appelant de nouveau le générateur sur la profondeur 0 (puisqu'on est en fin de simulation), et sur un nouvel ensemble d'instances ;

iii. cet ensemble d'instances est constitué de l'instance qui complète la primitive temps-réel, de toutes les instances activées à la date 0, et de toutes les instances qui ne sont pas encore activées ;

iv. $Test$ n'est pas modifié, car nous ne sommes pas en présence d'un test ;

2. $\bigotimes_{t>0}(\phi,\ Test) = \{\Theta.\lambda,\ /\ \lambda \in \bigotimes_{t-1}(\phi,\ Test)\}$.
Si à un instant donné il n'existe plus aucune instance à traiter, et nous ne sommes pas en fin de simulation, nous introduisons un temps d'inactivité processeur dans l'ensemble des arbres déjà construits, et nous rappelons le constructeur à l'instant suivant.
Cette approche va permettre de compléter les arbres par des temps creux quand nous sommes en fin de construction.
L'ensemble $Test$ n'est pas modifié ;

3. si $d_1 > 0$, cela veut dire qu'aucune instance de tâche n'est réveillée.
Nous introduisons un temps d'inactivité processeur à l'ensemble des arbres déjà construits, et nous rappelons le constructeur à l'instant suivant.
$\bigotimes_t([T^{d_j}_{i_j,n_j}]_{1\leq j\leq m},\ Test) = \{\Theta.\lambda,\ /\ \lambda \in \bigotimes_{t-1}([T^{d_j-1}_{i_j,n_j}]_{1\leq j\leq m},\ Test)\}$.
L'ensemble $Test$ n'est pas modifié ;

4. nous supposons qu'il existe maintenant des instances réveillées.
Si $\exists\ j > 0$ tel que $d_1 = d_2 = \ldots = d_j = 0$, $d_{j+1} > 0$, alors :

   (a) si toutes ces instances commencent par une instruction déclarative ou une primitive temps-réel, nous pouvons réorganiser la liste de ces instances afin d'avoir[1] :
   $$\begin{cases} T_{i_l} = a_{i_l}.T'_{i_l}\ 1 \leq l \leq u\ :\ \textit{instances déclaratives} \\ T_{i_l} = a_{i_l}.T'_{i_l}\ u < l \leq j\ :\ \textit{primitives temps-réel} \end{cases}$$
   L'opérateur est dans ce cas défini par :
   $\bigotimes_t([T^{d_j}_{i_j,n_j}]_{1\leq j\leq m},\ Test) = \bigcup_{l=1}^{u}\{a_{i_l}.\lambda,\ tel\ que$
   $\lambda \in \bigotimes_{t-1}([T'^{0}_{i_l,n_l}, (T^0_{i_1,n_1},\ldots,T^0_{i_j,n_j})\backslash(T^0_{i_l,n_l}), T^{d_{j+1}-1}_{i_{j+1},n_{j+1}},\ldots,T^{d_m-1}_{i_m,n_m}],\ Test)\}$
   $\cup$
   $\bigcup_{l=u+1}^{j}\{a_{i_l}.\lambda,\ tel\ que$
   $\lambda \in \bigotimes_t([T'^{0}_{i_l,n_l}, (T^0_{i_1,n_1},\ldots,T^0_{i_j,n_j})\backslash(T^0_{i_l,n_l}), T^{d_{j+1}}_{i_{j+1},n_{j+1}},\ldots,T^{d_m}_{i_m,n_m}],\ Test)\}$.
   Nous construisons donc un ensemble d'arbres commençant par les instructions déclaratives et nous rappelons le constructeur en décrémentant la durée (puisque nous avons exécuté une instruction déclarative) sur un nouvel ensemble d'instances, dont nous avons précédemment expliqué la

[1] Notons que ce cas est contenu dans le cas 4b. Il est traité ici par soucis de clarté

constitution.
Puis nous construisons un ensemble d'arbres commençant par les primitives temps-réel, et nous rappelons le constructeur sans décrémenter la durée (puisque les durées d'exécutions des primitives temps-réel sont supposées nulles), sur un nouvel ensemble d'instances.
Dans ce nouvel ensemble d'instances, les dates d'activation des instances qui ne sont pas encore activées ne sont pas décrémentées, parce qu'il s'agit de l'exécution des primitives temps-réel de durées supposées nulles.
Enfin, l'ensemble $Test$ n'est pas modifié, car aucun test n'est considéré ;

(b) sinon, il existe des instances réveillées qui commencent par un test.
Soit $t_{i_l,k}^{n_{i_l}}$ ce test.
Il s'agit de ne construire que des arbres d'ordonnancement sémantiquement corrects.
Nous vérifions s'il existe un test appartenant à $Test$ qui empêche la construction de certaines branches.
La suite de la construction dépend des résultats obtenus avec $t_{i_l,k}^{n_{i_l},+}$ et $t_{i_l,k}^{n_{i_l},-}$.
Nous faisons un nouveau partitionnement de la liste des instances :

$$\begin{cases} T_{i_l} = t_{i_l,k}^{n_{i_l}}(T_{i_l}^+, T_{i_l}^-)\ 1 < l \leq a\ et\ \exists\ te \in Test\ tel\ que\ t_{i_l,k}^{n_{i_l},-}\ \mathcal{RIT}\ te \\ T_{i_l} = t_{i_l,k}^{n_{i_l}}(T_{i_l}^+, T_{i_l}^-)\ a < l \leq b\ et\ \exists\ te \in Test\ tel\ que\ t_{i_l,k}^{n_{i_l},+}\ \mathcal{RIT}\ te \\ T_{i_l} = t_{i_l,k}^{n_{i_l}}(T_{i_l}^+, T_{i_l}^-)\ b < l \leq c\ :\ \forall\ te \in Test\ t_{i_l,k}^{n_{i_l},-}\ \overline{\mathcal{RIT}}\ te\ et\ t_{i_l,k}^{n_{i_l},+}\ \overline{\mathcal{RIT}}\ te \\ T_{i_l} = a_{i_l}.T_{i_l}^{'}\ c < l \leq d\ :\ instances\ déclaratives \\ T_{i_l} = a_{i_l}.T_{i_l}^{'}\ d < l \leq e\ :\ primitives\ temps\text{-}réel \end{cases}$$

L'opérateur est dans ce cas défini par :
$\bigotimes_t([T_{i_j,n_j}^{d_j}]_{1\leq j\leq m},\ Test) = A_1 \cup A_2 \cup A_3 \cup A_4 \cup A_5$ où :

i. $A_1 = \bigcup_{l=1}^{a}\{t_{i_l,k}^{n_{i_l},+}\lambda^+,\ tel\ que\ \ \lambda^+ \in$
$\bigotimes_{t-1}([T_{i_l,n_l}^{'+0}, (T_{i_1,n_1}^0, \ldots, T_{i_u,n_u}^0) \setminus (T_{i_l,n_l}^0), T_{i_{j+1},n_{j+1}}^{d_{j+1}-1}, \ldots, T_{i_m,n_m}^{d_m-1}],$
$Test \cup \{t_{i_l,k}^{n_{i_l},+}\})\}$.
Nous construisons un nouvel ensemble d'arbres, où chaque arbre commence par le résultat positif du test, donc nous ne considérons pas les branches induites par le résultat négatif du test, parce qu'il y a incompatibilité avec un test déjà rencontré.
Nous ajoutons à l'ensemble $Test$ le résultat du test positif, car il est pris en compte.
Enfin, nous rappelons le constructeur sur un nouvel ensemble en décrémentant la durée car les tests sont de durée unitaire ;

ii. $A_2 = \bigcup_{l=a+1}^{b}\{t_{i_l,k}^{n_{i_l},-}\lambda^-,\ tel\ que\ \ \lambda^- \in$
$\bigotimes_{t-1}([T_{i_l,n_l}^{'-0}, (T_{i_1,n_1}^0, \ldots, T_{i_u,n_u}^0) \setminus (T_{i_l,n_l}^0), T_{i_{j+1},n_{j+1}}^{d_{j+1}-1}, \ldots, T_{i_m,n_m}^{d_m-1}],$
$Test \cup \{t_{i_l,k}^{n_{i_l},-}\})\}$.
Ce cas est pareil au cas 4(b)i, avec pour unique différence qu'il porte sur le résultat négatif du test ;

iii. $A_3 = \bigcup_{l=b+1}^{c}\{(t_{i_l,k}^{n_{i_l},+}\lambda^+, t_{i_l,k}^{n_{i_l},-}\lambda^-)$ *tel que* $\lambda^+ \in$
$\bigotimes_{t-1}([T_{i_l,n_l}'^{+0}, (T_{i_1,n_1}^0, \ldots, T_{i_u,n_u}^0) \setminus (T_{i_l,n_l}^0), T_{i_{j+1},n_{j+1}}^{d_{j+1}-1}, \ldots, T_{i_m,n_m}^{d_m-1}]$,
$Test \cup \{t_{i_l,k}^{n_{i_l},+}\})$,
$\lambda^- \in$
$\bigotimes_{t-1}([T_{i_l,n_l}'^{-0}, (T_{i_1,n_1}^0, \ldots, T_{i_u,n_u}^0) \setminus (T_{i_l,n_l}^0), T_{i_{j+1},n_{j+1}}^{d_{j+1}-1}, \ldots, T_{i_m,n_m}^{d_m-1}]$,
$Test \cup \{t_{i_l,k}^{n_{i_l},-}\})\}$.
Dans ce cas, comme il n'y a aucune incompatibilité avec un test déjà rencontré, les deux branches induites par les résultats des tests positifs et négatifs sont considérées.
Nous construisons donc un nouvel ensemble d'arbres, où chaque arbre a deux branches, indexées respectivement par les résultats des tests positifs et négatifs, et le constructeur est rappelé respectivement comme dans les cas 4(b)i et 4(b)ii ;

iv. $A_4 = \bigcup_{l=c+1}^{d}\{a_{i_l}.\lambda,$ *tel que* $\lambda \in$
$\bigotimes_{t-1}([T_{i_l,n_l}'^{0}, (T_{i_1,n_1}^0, \ldots, T_{i_j,n_j}^0) \setminus (T_{i_l,n_l}^0), T_{i_{j+1},n_{j+1}}^{d_{j+1}-1}, \ldots, T_{i_m,n_m}^{d_m-1}],\ Test)\}$.
Ce cas a déjà été traité au point 4a ;

v. $A_5 = \bigcup_{l=d+1}^{e}\{a_{i_l}.\lambda,$ *tel que* $\lambda \in$
$\bigotimes_{t}([T_{i_l,n_l}'^{0}, (T_{i_1,n_1}^0, \ldots, T_{i_j,n_j}^0) \setminus (T_{i_l,n_l}^0), T_{i_{j+1},n_{j+1}}^{d_{j+1}}, \ldots, T_{i_m,n_m}^{d_m}],\ Test)\}$.
Ce cas a déjà été traité au point 4a.

Nous définissons ensuite l'opérateur de génération des arbres d'ordonnancement de profondeur T à partir d'un ensemble d'applications temps-réel multi-tâches :

$\tilde{\bigotimes}_T$:{Ensemble des applications temps-réel multi-tâches} $\rightarrow$ {Ensemble des arbres d'ordonnancement sur $[0, T]$} :
$\tilde{\bigotimes}_T([T_1, \ldots, T_n]) \equiv \bigotimes_T([T_{i_j,n_j}^{d_j}]_{1 \leq j \leq m},\ \emptyset)$.

Notons que cet opérateur ne construit que des arbres *conservatifs* (c'est à dire que le processeur est inactif quand il n'y a aucune tâche prête à être exécutée).
Cela n'est pas toujours avantageux, dans un contexte comme c'est le notre où les tâches sont dépendantes (voir exemple 5.4.1.1).
De plus, dans ce type d'ordonnancement, tous les temps creux se retrouvent en fin de simulation.
Ce qui n'est pas intéressant si on veut les exploiter, par exemple pour la gestion des pannes ou pour une gestion de tâches apériodiques en arrière plan.
Pour pallier ce problème, nous modifions le générateur en introduisant dès le début la tâche oisive qui est une tâche qui ne possède que des instructions déclaratives, et elle est utilisée comme telle pendant la génération des arbres d'ordonnancement.
L'opérateur $\tilde{\bigotimes}_T$ génère de façon *brute* et *exponentielle* (en fonction du nombre de tâches en entrée) un ensemble d'arbres d'ordonnancement sémantiquement valides.
Il faut ensuite utiliser les filtres présentés section 6.3 pour ne retenir que ceux qui sont valides suivant tous les autres aspects.

Notons enfin qu'en l'absence d'instructions conditionnelles dans le corps des tâches constituant l'application, l'ensemble généré par notre opérateur sera un ensemble de séquences, puisque le cas 4b ne sera pas considéré.

6.4.3 Mise en œuvre

Nous avons implémenté ce générateur, et il est actuellement en phase de tests. L'outil *GENTREE* dont le code est donné en annexe .2, permet d'obtenir l'ensemble des arbres d'ordonnancement d'une application donnée.

Bilan

Le travail effectué jusqu'à présent dans cette thèse est résumé figure 6.4.

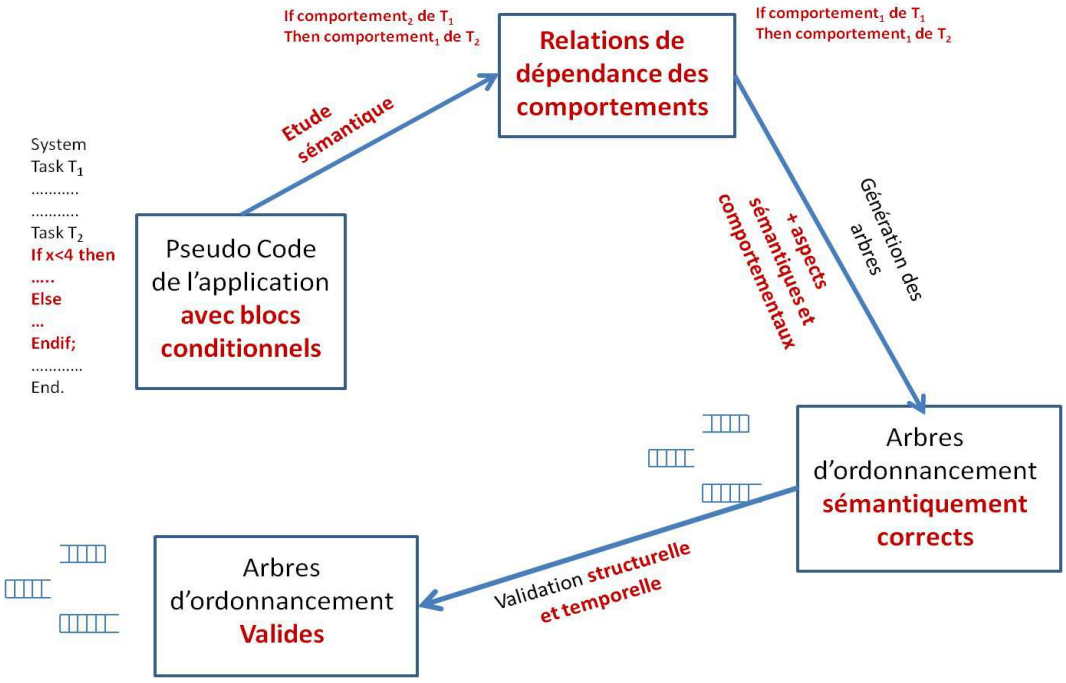


Figure 6.4 – Obtention brute de tous les arbres d'ordonnancement

Pour modéliser le comportement des applications temps-réel pendant leur exécution, nous avons proposé d'utiliser, non plus les séquences d'ordonnancement, mais les arbres d'ordonnancement.
Ils ont été présentés dans ce chapitre.
Ils permettent de représenter finement les différents comportements des tâches, mais surtout de tenir compte des éventuelles corrélations entre elles.
Enfin, nous avons proposé une approche d'obtention de ces arbres d'ordonnancement.

L'approche utilisée est hautement combinatoire, le nombre d'arbres potentiels étant rapidement élevé.
Cette approche reste donc théorique, car elle ne peut être industrialisée, et ne s'applique que pour de très petites applications.
D'autre part, le principe consiste à construire tous les arbres, puis à en éliminer ceux qui ne sont pas valides.
Un outil plus efficace s'avère nécessaire.
Nous le présentons au chapitre 8.
Il s'agit d'une approche orientée modèle basée sur les réseaux de Petri, qui permettra d'obtenir les arbres recherchés et dont la complexité peut être réduite grâce à de bonnes heuristiques.
De plus, elle intégrera les contraintes de validité dans le processus de construction, ce qui permettra d'obtenir directement les arbres d'ordonnancement valides.

Pour l'instant, dans le chapitre 7 qui suit, nous allons valider notre approche, en comparant de façon générale l'approche par arbres d'ordonnancement et l'approche par encapsulation.
Il s'agira de poser les cadres et limites d'utilisation de ces deux notions.

Chapitre 7

Ordonnancement linéaire versus ordonnancement arborescent

"*Une condition essentielle de l'hypothèse (scientifique) c'est qu'elle soit aussi probable que possible*"
Claude Bernard

Sommaire

Afin d'intégrer les éléments sémantiques (relations intra-tâches et inter-tâches) pendant l'analyse d'ordonnançabilité, nous avons proposé d'utiliser les arbres d'ordonnancement, en lieu et place des séquences d'ordonnancement obtenues lorsque l'on encapsule les blocs conditionnels.
Nous nous intéressons dans ce chapitre à la comparaison entre ces deux approches de modélisation de comportements des applications temps-réel.
De façon plus précise, nous voulons montrer que l'analyse arborescente raffine l'analyse par encapsulation, sans remettre en cause les résultats positifs qu'elle produit.
Nous commençons par montrer que lorsque les tâches de l'application ne contiennent pas d'instructions conditionnelles, les résultats obtenus par les deux approches sont semblables : l'ensemble des arbres d'ordonnancement (valides) est le même que celui des séquences d'ordonnancement (valides).
Ensuite, nous travaillons sur des tâches comportant des instructions conditionnelles. Nous montrons que les approches de validation basées sur l'encapsulation sont parfois pessimistes, et peuvent conduire à des conclusions d'ordonnançabilité erronées. De façon plus précise, certaines applications peuvent être déclarées non ordonnançables avec les approches classiques basées sur les séquences et l'encapsulation, alors qu'en réalité elles sont ordonnançables lorsque l'on adopte l'approche arborescente. L'approche arborescente, plus détaillée que l'approche linéaire est donc parfois nécessaire.
Par contre, dans le cas où l'approche linéaire donne des résultats positifs sur l'ordonnançabilité de l'application, ceux ci ne sont pas remis en cause par l'approche

arborescente.
Nous montrons donc que, sous réserve que les contraintes structurelles du système soient respectées, s'il est ordonnançable par l'approche par encapsulation, alors il est ordonnançable selon l'approche basée sur les arbres d'ordonnancement.

7.1 Notations

Soit $S = \{T_1, \ldots, T_n\}$ une application temps-réel constituée de n tâches périodiques, pouvant chacune contenir des instructions conditionnelles, données suivant le contexte de l'étude établi section 1.2.1.
Les tâches peuvent donc être représentées sous leur forme arborescente (voir chapitre 4).
Soit $\overline{S} = \{\overline{T_1}, \ldots, \overline{T_n}\}$ le système déduit de S, où $\overline{T_i} = Lineaire(T_i)$, $Lineaire$ étant l'opérateur défini chapitre 4 qui permet le passage d'une tâche de sa représentation arborescente vers sa représentation linéaire.
Nous appellerons $\overline{S}$ le système linéaire de S, et nous noterons $\overline{U}$ le facteur de charge de l'application calculé en utilisant $\overline{S}$.
Par la suite, les séquences d'ordonnancement seront utilisées pour analyser $\overline{S}$, et les arbres d'ordonnancement pour l'étude de S.
Nous notons $\mathcal{SEQ}(S)$ l'ensemble des séquences d'ordonnancement de S et $\mathcal{ARB}(S)$ l'ensemble des arbres d'ordonnancement valides de S.

7.2 Tâches sans instructions conditionnelles

Les tâches considérées dans cette section ne comportent pas de tests conditionnels.

Théorème 7.2.1 Égalité des ensembles d'ordonnancement
Pour une application temps-réel dont les tâches ne possèdent pas d'instructions conditionnelles, l'ensemble des arbres d'ordonnancement valides est égal à l'ensemble des séquences d'ordonnancement.

Preuve 7.2.1 Égalité des ensembles d'ordonnancement
Soit $S = \{T_1, \ldots, T_n\}$ une application temps-réel constituée de tâches T_i sans instructions conditionnelles.
Nous avons donc, $\forall\ i \in 1 \ldots n$, $\overline{T_i} = Lineaire(T_i) = T_i$. Par conséquent, $\overline{S} = S$.
Nous voulons montrer que, dans ce contexte, $\mathcal{SEQ}(S) = \mathcal{ARB}(S)$.
Nous allons montrer les deux sens de l'inclusion :

1. $\mathcal{SEQ}(S) \subseteq \mathcal{ARB}(S)$
   Soit $\mathcal{S} \in \mathcal{SEQ}(S)$. $\mathcal{S} = s_1 \ldots . s_l$, avec $s_i \in \alpha\beta$, et $\mathcal{S}$ vérifie les propriétés de la définition 6.1.1 (qui correspond à la définition d'une séquence d'ordonnancement).
   Nous construisons l'arbre de la figure 7.1.
   La figure 7.1 est la représentation de S sous la forme d'un arbre : les étiquettes

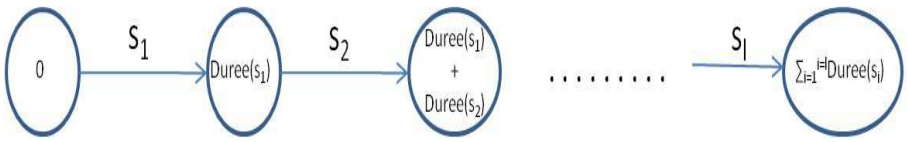


Figure 7.1 – Transformation d'une séquence d'ordonnancement en un arbre d'ordonnancement

sont les éléments de $\alpha\beta$ qui constituent $\mathcal{S}$, et la clé des nœuds est une suite croissante des durées d'éléments de $\mathcal{S}$.
L'arbre ainsi construit vérifie les propriétés suivantes de la définition 6.2.1 des arbres d'ordonnancement :

(a) $cle(racine) = 0$;

(b) $cle(nœud\ terminal) = \sum_{i=1}^{i=l} Duree(s_i) = T$ par définition de la séquence (définition 6.1.1) ;

(c) le point 3 de la définition 6.2.1 est vérifié par construction ;

(d) enfin, le point 4 de la définition 6.2.1 est un cas impossible, car l'arbre construit à partir de $\mathcal{S}$ n'a pas d'instructions conditionnelles.

Par conséquent, $\mathcal{S} \in \mathcal{ABR}(S)$.
Il reste à justifier que $\mathcal{S}$ est valide :

(a) la validité structurelle de $\mathcal{S}$ se déduit de la validité structurelle de la séquence d'ordonnancement, comme énoncée section 6.1.2 ;

(b) les validités comportementale et sémantique sont immédiates, car l'arbre construit ne possède pas de tests conditionnels ;

(c) enfin, la validité temporelle se déduit des points 2, 3 et 4 de la définition 6.1.1.

$\mathcal{S}$ est donc valide. □

2. $\mathcal{ARB}(S) \subseteq \mathcal{SEQ}(S)$
Soit $\mathcal{A} \in \mathcal{ARB}(S)$.
$\mathcal{A}$ est un arbre dont les liens sont étiquetés par l'alphabet $\alpha\gamma$.
$\mathcal{A}$ peut être représenté comme à la figure 7.2, car $\nexists\ \eta$ tel que $||fils(\eta)|| = 2$.

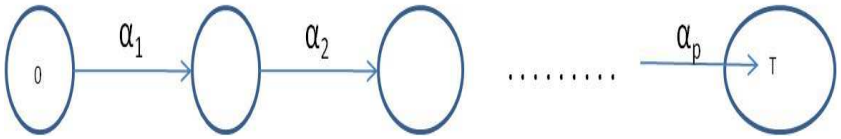


Figure 7.2 – Un arbre d'ordonnancement sans nœud double

Dans la figure 7.2, les nœuds de l'arbre vérifient, par définition, les points 1, 2 et 3 de la définition 6.2.1 des arbres d'ordonnancement.

Nous construisons la séquence $\alpha_1.\alpha_2.\ldots.\alpha_p$ qui correspond à cette représentation.
Elle vérifie les points 1 et 2 de la définition 6.1.1 des séquences d'ordonnancement :

(a) $\sum_{i=1}^{i=l} Duree(s_i)$ correspond par définition à la profondeur du nœud terminal, qui est T ;

(b) les points 2, 3 et 4 sont déduits des points 1 et 2 de la caractérisation des arbres temporellement valides (section 6.3.3).

Nous pouvons donc conclure que $\mathcal{A} \in \mathcal{SEQ}(S)$. □

Les deux sens de l'inclusion étant prouvés, $\mathcal{SEQ}(S) = \mathcal{ARB}(S)$, dans le cas où les tâches ne possèdent pas d'instructions conditionnelles.

7.3 Tâches avec instructions conditionnelles

Dans toute cette section, les applications considérées ont au moins une tâche qui possède au moins un test conditionnel.
Notre objectif est ici de montrer :

1. d'une part que l'approche arborescente est nécessaire, car on peut avoir un résultat négatif d'ordonnançabilité par l'approche linéaire, mais un résultat positif d'ordonnançabilité par l'approche arborescente ;
2. d'autre part qu'un résultat d'ordonnançabilité obtenu par l'approche linéaire (par encapsulation) reste valide par l'approche arborescente.

7.3.1 Quand l'application n'est pas classiquement ordonnançable

Dans le cas où l'application temps-réel n'est pas ordonnançable suivant les approches classiques, nous avons établi le résultat 7.3.1.

Résultat 7.3.1 Soit une application temps-réel S.
Si $\overline{S}$ n'est pas ordonnançable, S peut être ordonnançable ou pas.

Nous allons justifier le résultat 7.3.1 sur deux exemples :

1. dans l'exemple 7.3.1.1, nous donnons une application temps-réel qui n'est pas ordonnançable par l'approche par encapsulation, mais qui l'est par l'approche arborescente ;
2. dans l'exemple 7.3.1.2, nous considérons une application temps-réel qui n'est pas ordonnançable par l'approche par encapsulation, et qui n'est non plus pas ordonnançable par l'approche arborescente.

Exemple 7.3.1.1 $\overline{S}$ n'est pas ordonnançable et S est ordonnançable
Soit le système des essuie-glaces de l'exemple 1.2.2.1 dont le code est donné en annexe .1 et les représentations linéaire et arborescente des tâches le constituant sont données chapitre 4.

Pour le système $\overline{S}$, comme le facteur de charge $\overline{U} = \frac{2}{11} + \frac{6}{11} + \frac{4}{11} = \frac{12}{11} > 1$, nous pouvons conclure ([Buttazo 1997]) que le système des essuie-glaces n'est pas ordonnançable.
Cela veut dire qu'on ne peut pas trouver de séquence d'ordonnancement valide à partir des tâches constituant ce système.
De façon concrète, cela veut dire que, sur la base de cette analyse, les concepteurs n'implémenteront pas le système dans le véhicule, parce qu'ils ne peuvent pas garantir qu'il s'activera dans les délais voulus en cas de pluie.
Pourtant, dans la pratique, le système fonctionne correctement.
En effet, l'arbre d'ordonnancement donné figure 6.3 est un arbre valide.
La situation conduisant à un dépassement d'échéance est éliminée par l'analyse sémantique : elle correspond à l'enchainement de comportements incompatibles des tâches T_2 et T_3 (induites par 2 valeurs différentes du paramètre *contact*).
Le système est donc ordonnançable en utilisant l'approche arborescente.
De façon concrète, cela veut dire que le système des essuie-glaces peut être implémenté en toute sécurité dans le véhicule.
Le système est en réalité ordonnançable, bien qu'il ne le soit pas par l'approche linéaire par encapsulation.

Exemple 7.3.1.2 $\overline{S}$ n'est pas ordonnançable et S n'est pas ordonnançable
Soit S un système de 2 tâches T_1 et T_2 dont une représentation arborescente est donnée figure 7.3.

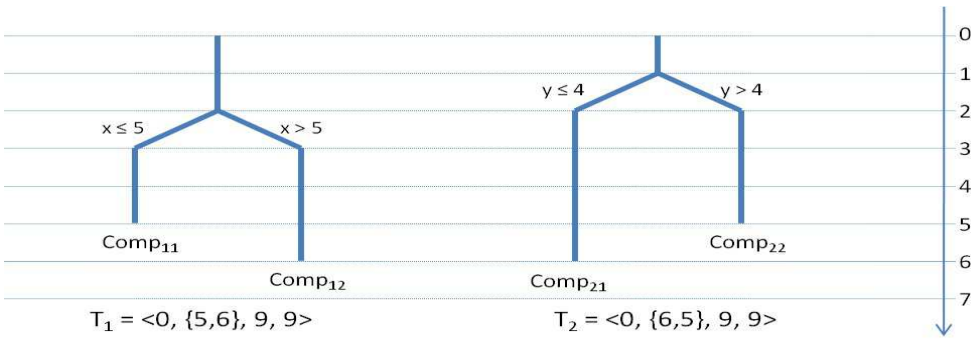


Figure 7.3 – Une application non ordonnançable quelque soit l'approche considérée

Une analyse de l'application en utilisant l'approche linéaire nous permet de conclure que l'application $\overline{S}$ n'est pas ordonnançable.
En effet, le facteur de charge $\overline{U} = \frac{6}{9} + \frac{6}{9} = \frac{12}{9} > 1$.
On ne peut donc pas trouver de séquence d'ordonnancement (valide) (au sens de la définition 6.1.1).
Nous considérons maintenant l'approche arborescente qui prend explicitement en compte les instructions conditionnelles.
Les tests étant indépendants (ils ne sont pas incompatibles), l'ensemble des combinaisons des comportements possibles est :

$\{(Comp_{11}, Comp_{21}),\ (Comp_{11}, Comp_{22}),\ (Comp_{12}, Comp_{21}),\ (Comp_{12}, Comp_{22})\}$.
Quelque soit la combinaison choisie, le facteur de charge correspondant est strictement plus grand que 1.
L'application n'est donc pas localement ordonnançable (voir section 3.4.3).
Par conséquent, elle n'est pas globalement ordonnançable (voir section 3.4.3).
On ne peut pas trouver un arbre d'ordonnancement qui ordonnance cette application.
L'application n'est donc ordonnançable ni en utilisant l'approche linéaire, ni en utilisant l'approche arborescente.

L'approche classique linéaire n'est donc pas suffisante pour conclure à la non ordonnançabilité des applications temps-réel.
Cette constatation justifie le besoin de disposer d'outils pour déployer l'approche arborescente.
Les outils que nous proposons sont décrits dans le chapitre 8.

7.3.2 Quand l'application est classiquement ordonnançable

Nous nous intéressons dans cette section aux applications qui ont été déclarées ordonnançables en utilisant l'approche linéaire.
Notre objectif est :

1. d'une part, de montrer qu'elles restent ordonnançables quand nous utilisons l'approche arborescente ;
2. et d'autre part, de montrer que ce résultat n'est valide que si les hypothèses structurelles sont vérifiées.

7.3.2.1 Préservation de l'ordonnançabilité linéaire

Nous avons vu (résultat 7.3.1), que si une application linéaire n'était pas ordonnançable, alors l'application arborescente correspondante pouvait être ordonnançable ou pas.
Nous montrons dans cette section que si une application temps-réel est déclarée ordonnançable en utilisant l'approche linéaire par encapsulation, alors elle reste ordonnançable en utilisant l'approche arborescente.

Théorème 7.3.1 Préservation de l'ordonnançabilité linéaire
Soit S une application temps-réel structurellement valide (voir définition 5.3.1).
Si $\overline{S}$ est ordonnançable, alors S est ordonnançable.

Preuve 7.3.1 Préservation de l'ordonnançabilité linéaire
Soit $\mathcal{S}$ une séquence d'ordonnancement (valide) de $\overline{S}$.
Notre objectif est de construire à partir de $\mathcal{S}$ un arbre d'ordonnancement valide de S.
Nous commençons par définir certaines notions que nous utiliserons pendant la démonstration.

Définition 7.3.1 Niveau d'un test dans une tâche
Soit $T_i = B_{i1}.B_{i2}.\ldots.B_{in_i}$, une tâche (voir définition 4.1.1), et $t \in \mathcal{T}est_i$.
Nous définissons le **niveau d'un test** dans une tâche, par la fonction $Niveau$ suivante :

$$\left\{\begin{array}{l} Niveau : \mathcal{T}est_i\ X\ Ensemble\ des\ tâches \bigcup Ensemble\ de\ blocs\ généraux \rightarrow \mathbb{N} \\ tel\ que : \\ Si\ \exists\ j\ tel\ que\ B_{ij}\ =\ If\ t\ then\ SB_{ij}^{+}\ else\ SB_{ij}^{-}\ endif \\ alors\ Niveau(t,\ T_i)\ =\ 1 \\ Sinon \\ Si\ t\ apparait\ dans\ SB_{ij}^{+}\ (\ SB_{ij}^{-}), \\ alors\ Niveau(t,\ T_i)\ =\ 1\ +\ Niveau(t,\ SB_{ij}^{+})\ (Niveau(t,\ SB_{ij}^{-})) \end{array}\right.$$

Pour illustration, en considérant la tâche de la figure 4.11, nous avons :

1. $Niveau(t_1, T) = 1$;
2. $Niveau(t_2, T) = 2$.

Définition 7.3.2 Date d'un test dans une tâche
Nous définissons la **date d'un test** t dans une tâche T, et nous notons $Date(t,\ T)$, son instant d'apparition au plus tard dans la tâche.
De façon formelle, nous avons :

$$\left\{\begin{array}{l} Date : \mathcal{T}est_i\ X\ Ensemble\ des\ tâches \bigcup Ensemble\ de\ blocs\ généraux \rightarrow \mathbb{N} \\ tel\ que : \\ Si\ Niveau(t,\ T_i)\ =\ 1 \\ \quad Si\ \exists\ j\ tel\ que\ B_{ij}\ =\ If\ t\ then\ SB_{ij}^{+}\ else\ SB_{ij}^{-}\ endif \\ \quad alors\ Date(t,\ T_i)\ =\ \sum_{p=1}^{j-1} duree(Lineaire(B_{ip})) \\ Sinon \\ \quad Si\ t\ apparait\ dans\ SB_{ij}^{+}\ (\ SB_{ij}^{-}), \\ \quad alors\ Date(t,\ T_i)\ =\ \sum_{p=1}^{j-1} duree(Lineaire(B_{ip}))\ +\ Date(t,\ SB_{ip}^{+})\ (Date(t,\ SB_{ip}^{-})) \end{array}\right.$$

Pour illustration, en considérant la tâche de la figure 4.11, nous avons :

1. $Date(t_1, T) = 1$;
2. $Date(t_2, T) = 5$.

Nous nous intéressons maintenant à la preuve effective du théorème 7.3.1.
Pour simplifier la présentation, nous supposons que les tâches de l'application considérée sont à départs simultanés.
Cette hypothèse pourra se généraliser par la suite en suivant le même principe.
Soit donc $\mathcal{S}$ une séquence d'ordonnancement (valide) de $\overline{S}$, et $\mathcal{A}$ l'arbre d'ordonnancement que nous allons construire.
Nous initialisons $\mathcal{A}$ par $\mathcal{S}$.
Nous avons donc au début $\mathcal{A} = \mathcal{S}$.

1. Pour $i \in 1, \ldots, n$,
   si T_i comporte des instructions conditionnelles,
   soit L_1 la liste des tests de *Niveau* 1 rangés par ordre croissant de *Date*
   (a) pour chaque test t de L_1,
   pour chaque branche de $\mathcal{A}$,
   pour chaque instance
      i. nous marquons l'instruction a_i située à distance *Date* de l'activation de l'instance (c'est à dire que nous marquons les nœuds) :
      pour cela, nous avons un compteur initialisé à 0.
      Nous parcourons la branche de l'instance.
      Nous incrémentons le compteur chaque fois que nous rencontrons un lien d'étiquette a_i.
      Quand le compteur est égal à *Date*, nous marquons le prochain nœud rencontré d'une part, et d'autre part, nous remplaçons le lien d'étiquette a_i par l'étiquette t.

   Puis, nous remplaçons chaque nœud marqué comme illustré figure 7.4 et nous faisons pareil dans chacun des sous arbres.

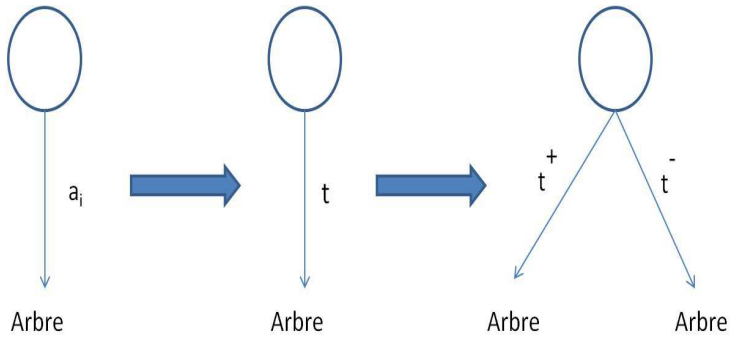


Figure 7.4 – Transformation d'un nœud simple en un nœud test

   Ensuite, pour $t \in L_1$,
   (a) nous construisons $L_2^+(t)$.
   Il s'agit de la liste des triplets (t_1, 2, $date\_depuis\_t^+$).
   À partir de chaque nœud de la figure 7.5, nous parcourons les branches de $\mathcal{A}^+$, nous avançons de $date\_depuis\_t^+$, et nous remplaçons le nœud rencontré, qui est un nœud simple, par un nœud test, comme illustré figure 7.6, et t_1 apparait dans le sous arbre de la tâche T_i induit par t^+.
   (b) puis nous faisons pareil pour $L_2^-(t)$, pour les branches de $\mathcal{A}^-$.

   Ensuite nous passons au *Niveau* 3 ....
   Nous procédons ainsi jusqu'à couvrir tous les niveaux existants.

Quand nous avons traité tous les niveaux de toutes les tâches, nous avons un arbre où les tests ont été placés.

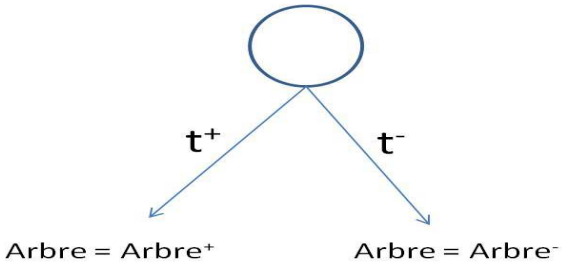


Figure 7.5 – Nœud de départ pour le *Niveau* 2

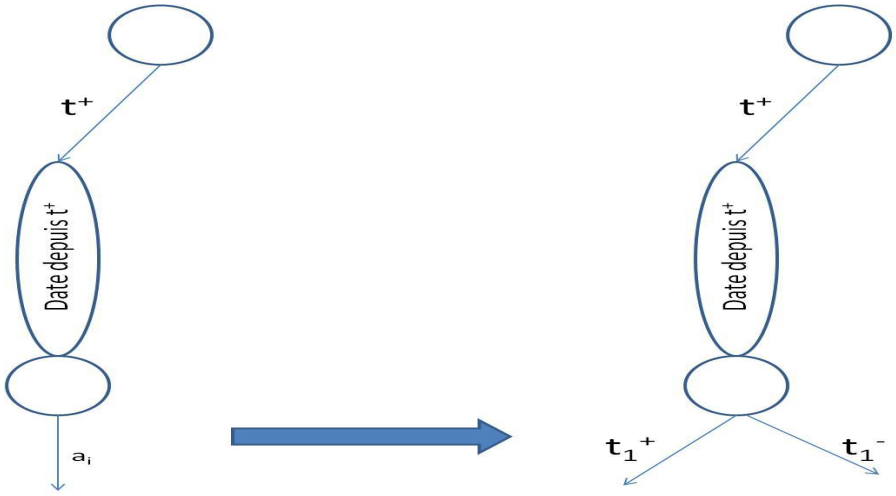


Figure 7.6 – Processus de transformation pour les nœuds de *Niveau* 2

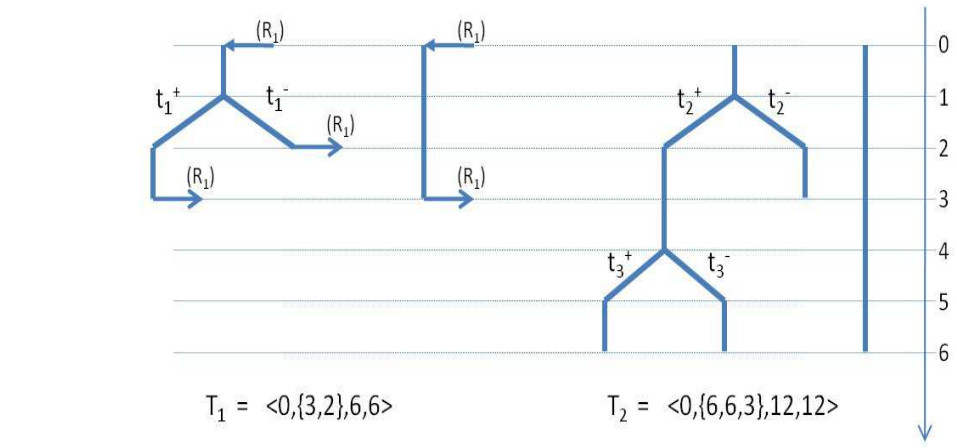
Toutes les échéances ont été respectées le long de chacune des branches, puisque aucune date de fin de tâche n'a été modifiée.
Ceci garantit la validité temporelle de l'arbre.
Chaque branche de l'arbre correspond à un comportement de l'application.
Ce qui garantit la validité comportementale.
Pour prendre en compte l'aspect sémantique pendant la construction de l'arbre, nous modifions le processus de transformation de la figure 7.6, à partir du test t_1.
Nous ne créons plus automatiquement les deux comportements relatifs à t_1^+ et t_1^-, mais uniquement les comportements provenant des tests compatibles avec les tests qui précèdent.
L'arbre obtenu est donc sémantiquement valide.
Il faut enfin gérer la validité structurelle :

1. nous effaçons le long de chaque branche les instructions a_i en trop, ainsi que les primitives temps-réel qui ne correspondent pas aux comportements sélectionnés ;
2. ensuite, éventuellement, nous réduisons les sections critiques : en effet, les sections critiques du modèle linéaire ont été obtenues par fusion des sections critiques des différentes branches de la tâche.
   Nous remettons donc les sections critiques initiales, qui sont plus petites.
   De ce fait, nous n'engendrons pas de problèmes.
   Si la ressource est prise un peu plus tard, elle sera toujours libre. Et on peut la libérer plus tôt, jamais plus tard ;
3. en ce qui concerne les envois de messages, on peut être amené à les avancer, ce qui ne posera pas de problèmes pour les tâches réceptrices ;
4. enfin, les réceptions peuvent être retardées.
   On est sûr que les messages auront été préalablement émis.

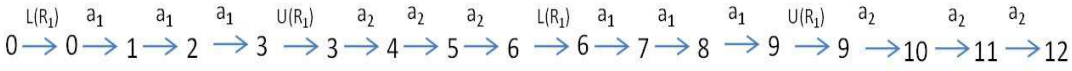
En conclusion, tous ces ré-agencements pour l'aspect structurel ne produiront pas de blocages, et les instructions de fin de tâche peuvent être avancées, jamais retardées.
Donc les échéances restent vérifiées.
Nous avons donc construit à partir de $\mathcal{S}$ un arbre d'ordonnancement valide. □

Nous allons illustrer la preuve 7.3.1 sur l'exemple 7.3.2.3.
Pour cela, nous allons considérer une application temps-réel structurellement valide.
Dans un premier temps, nous allons déduire son modèle linéaire, puis donner une séquence d'ordonnancement (valide) de ce modèle.
Notre objectif est de dérouler la preuve 7.3.1 afin d'obtenir un arbre d'ordonnancement valide.

Exemple 7.3.2.3 Illustration de la preuve 7.3.1
Soit S une application temps-réel dont les représentations arborescente, linéaire et temporelle des tâches T_1 et T_2 (indépendantes) la constituant sont données figure 7.7(a), ainsi qu'une séquence d'ordonnancement (valide) figure 7.7(b).
Nous avons : $Niveau(t_1, T_1) = 1$, $Niveau(t_2, T_2) = 1$ et $Niveau(t_3, T_2) = 2$.



(a)



(b)

Figure 7.7 – (a) S : représentations arborescentes, linéaires et temporelles. (b) Une séquence d'ordonnancement (valide) de $\overline{S}$

De même, nous avons $Date(t_1, T_1) = 1$, $Date(t_2, T_2) = 1$ et $Date(t_3, T_2) = 4$.
Nous initialisons l'arbre $\mathcal{A}$ à $\mathcal{S}$.
Pour T_1,
pour chaque branche,
pour chaque instance de T_1,
nous marquons les nœuds correspondant à $Date = 1$, et nous obtenons l'étape 1 de la figure 7.8, puis nous déroulons l'algorithme pour obtenir les étapes 2 et 3 de la figure 7.8.

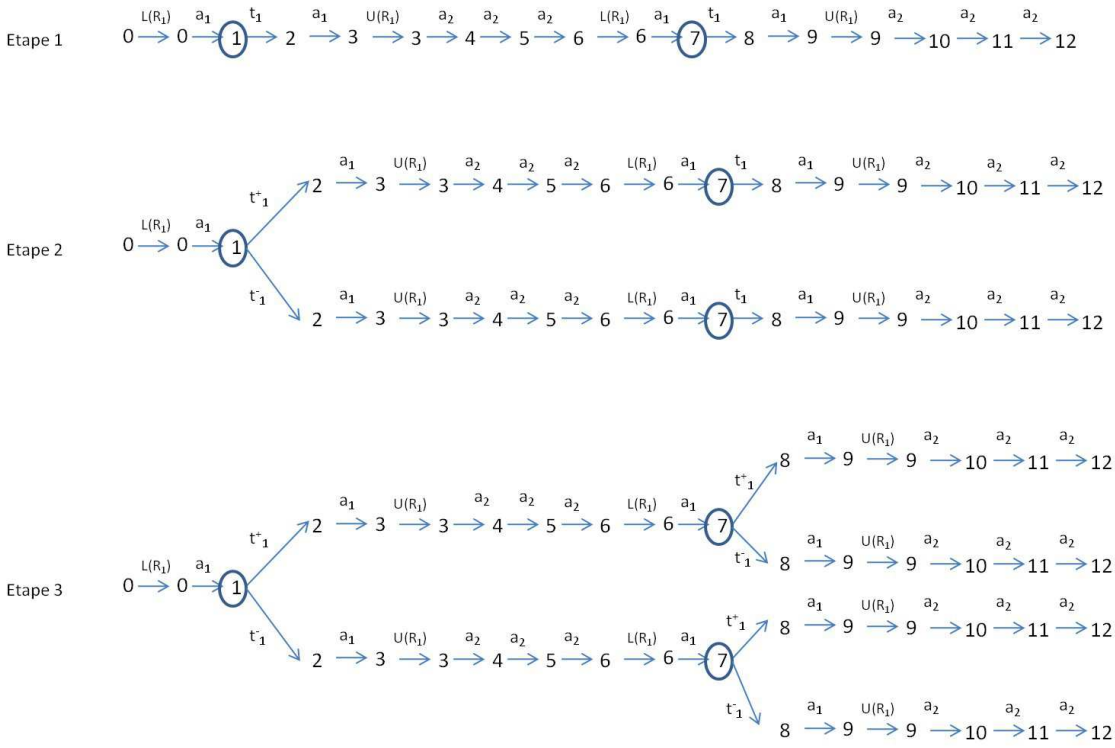


Figure 7.8 – Marquage des nœuds pour le test t_1 de T_1 et construction de l'arbre associé

Ensuite, $L_2^+(t_1) = L_2^-(t_1) = \emptyset$.
Nous arrêtons donc là l'algorithme pour T_1.
Pour T_2,
$L_1 = \{t_2\}$. Nous marquons les nœuds correspondants, puis nous déroulons l'algorithme, et nous obtenons l'étape 4 de la figure 7.9.
Puis, nous construisons $L_2^+(t_2) = \{t_3\}$ et $L_2^-(t_2) = \emptyset$.
Nous déroulons donc le cas t_3 à partir de t_2^+, et nous ne faisons aucune modifications à partir de t_2^-.
Nous obtenons donc l'arbre de l'étape 5 de la figure 7.10.
Enfin, nous effaçons les instructions en trop, nous réduisons les sections critiques.
Il n'y a pas de tests incompatibles, donc nous n'éliminons aucun sous arbre.

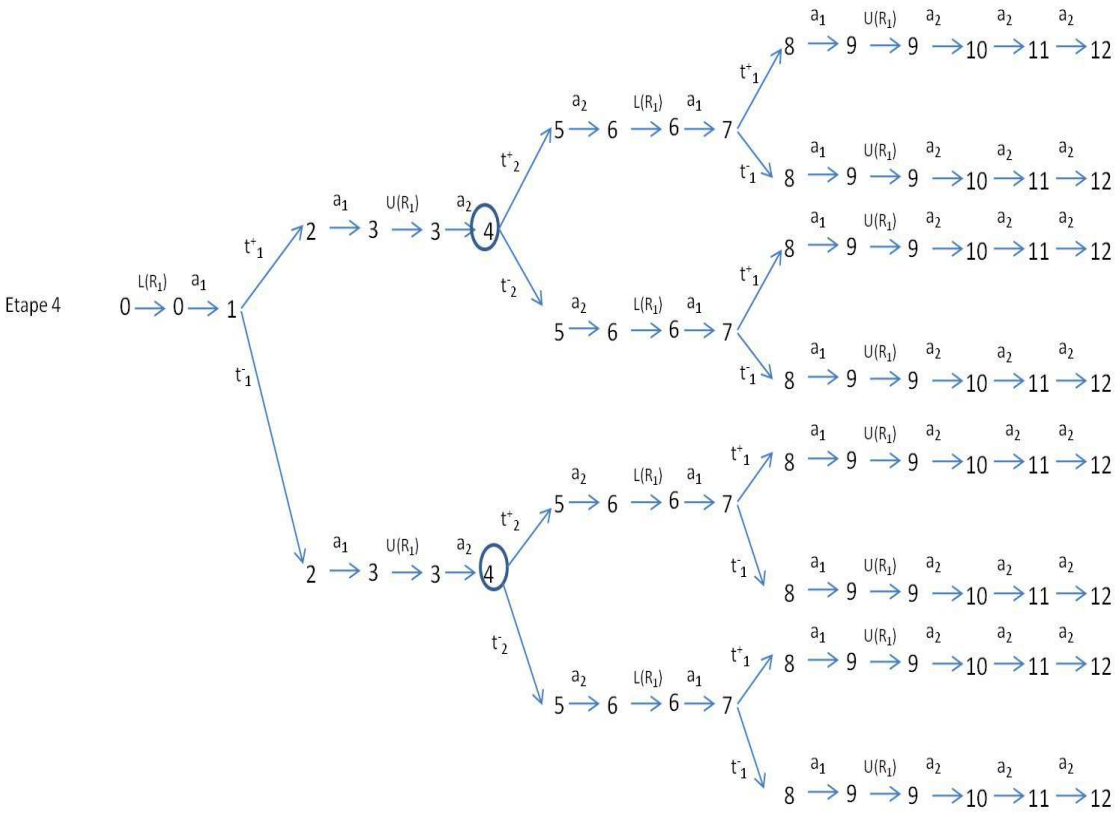


Figure 7.9 – Marquage des nœuds pour le test t_2 de T_2 et construction de l'arbre associé

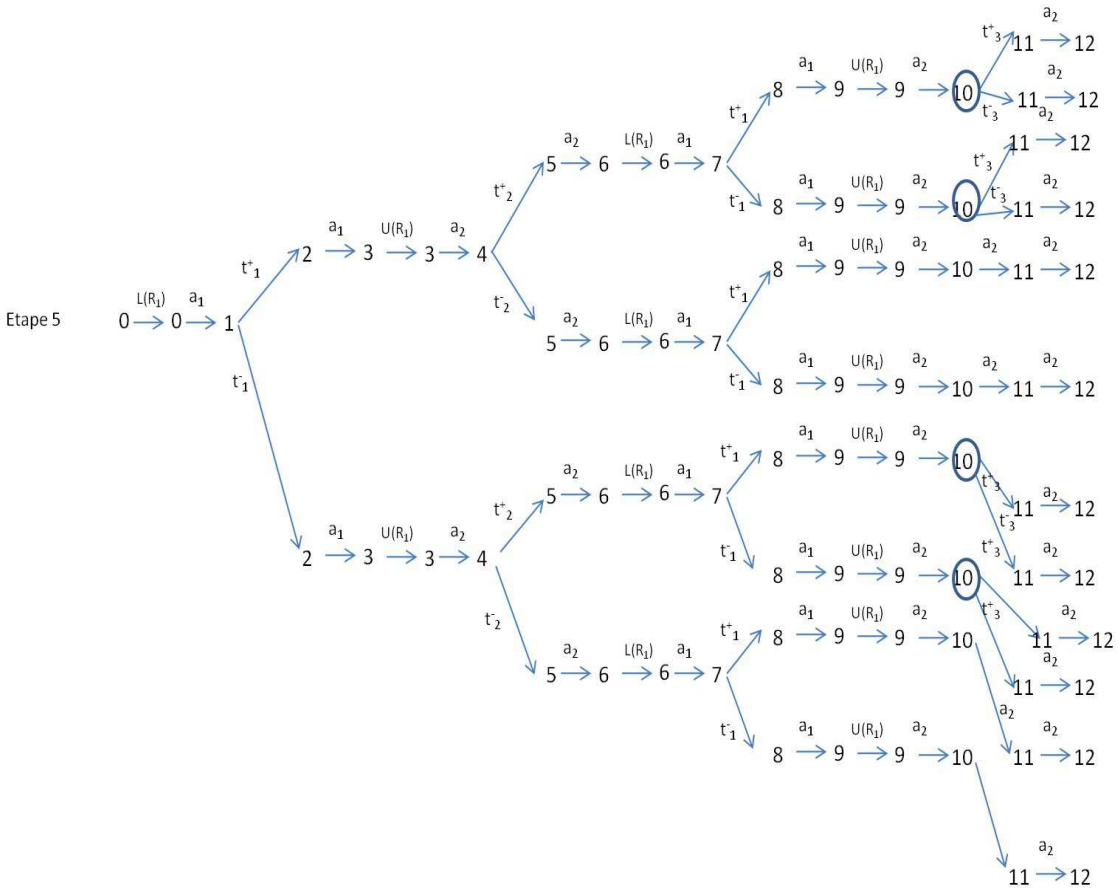


Figure 7.10 – Marquage des nœuds pour le test t_3 de T_2 et construction de l'arbre associé

De même, il n'y a pas d'envoi ou de réception de messages à compléter.
L'arbre final obtenu, qui est valide, est donné figure 7.11.

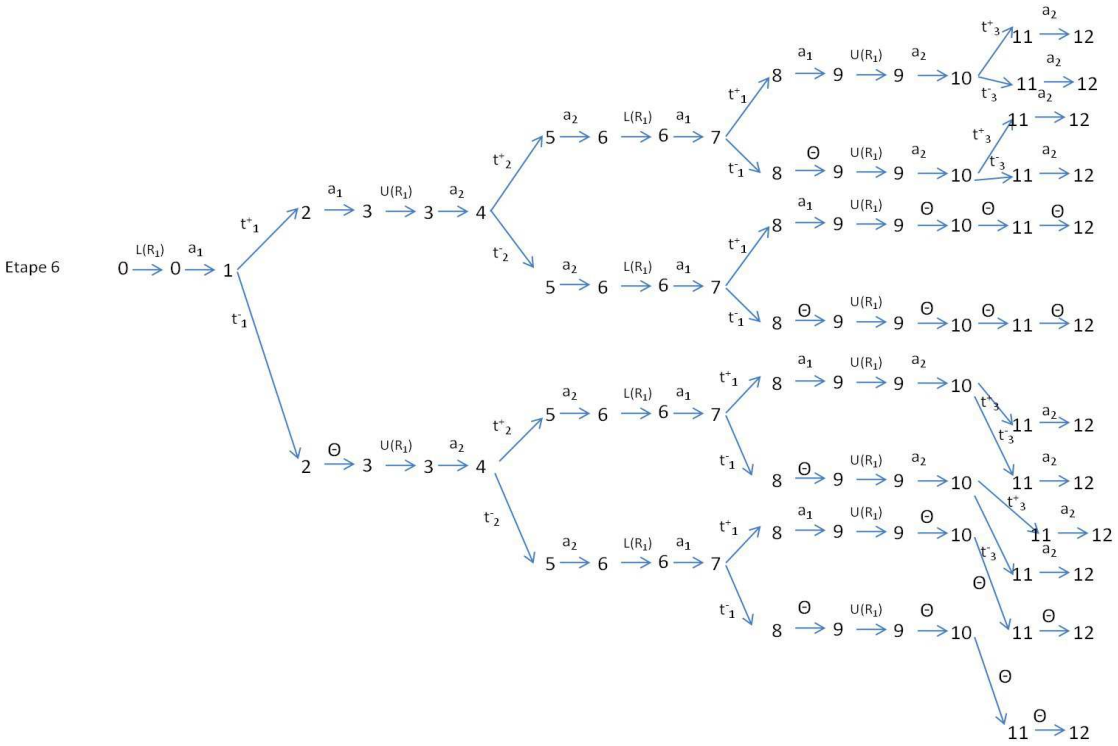


Figure 7.11 – L'arbre d'ordonnancement $\mathcal{A}$ valide obtenu à partir de $\mathcal{S}$. θ désigne un temps d'inactivité processeur

Dans ce cas, nous avons réduit les sections critiques dans certaines branches, ce qui explique le remplacement de certaines instructions a_i par θ.

Remarquons que l'arbre de la figure 7.11 a bien
$\prod_{i=1}^{n}(Compeff_i)^{\frac{P_i}{P}} = 2^2 * 3^1 = 12$ comportements, ce qui correspond au nombre maximum de comportements d'un arbre d'ordonnancement pour le système considéré.
Dans notre cas, il est égal a 12, parce que tous les comportements sont compatibles.

7.3.2.2 Validité structurelle

Dans le paragraphe précédent, nous avons montré que l'ordonnançabilité linéaire entrainait l'ordonnançabilité arborescente.
Nous avons pour cela supposé l'application structurellement valide.
Nous allons maintenant voir que cette hypothèse doit être vérifiée, faute de quoi le résultat précédent n'est plus valide.
En d'autres termes, la linéarisation peut masquer les incohérences structurelles.

Nous allons le justifier sur l'exemple 7.3.2.4.

Exemple 7.3.2.4 Respect de la validité structurelle obligatoire
Pour montrer que l'hypothèse de validité structurelle est fondamentale, nous considérons l'application temps-réel S de la figure 7.12, constituée de 2 tâches T_1 et T_2.

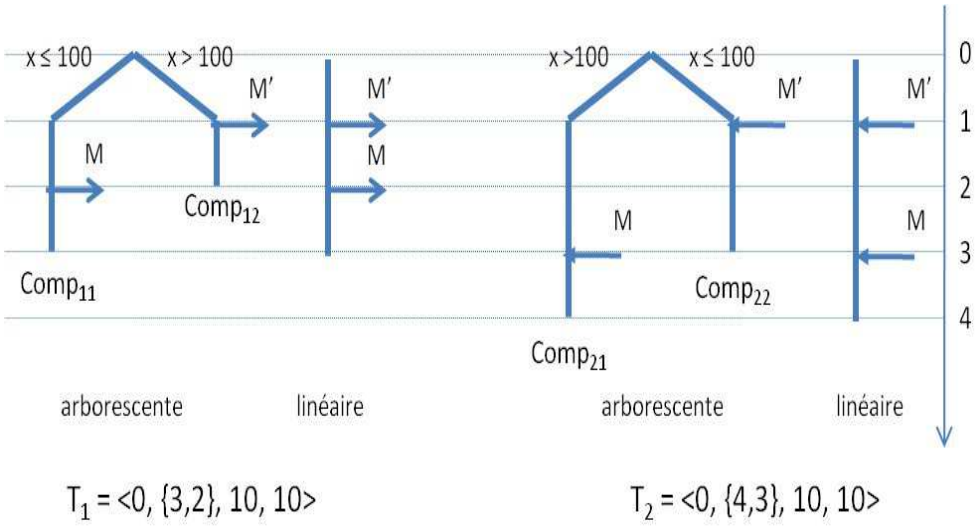


Figure 7.12 – Une application temps-réel utilisée pour montrer que le respect de la condition de validité structurelle est nécessaire pour une bonne conclusion d'ordonnançabilité

L'application S n'est pas structurellement valide :

1. les messages envoyés ne sont pas émis par tous les comportements ;
2. les messages reçus ne sont pas reçus par tous les comportements compatibles.

Nous allons maintenant analyser $\overline{S}$ et S.
En utilisant l'approche linéaire, $\overline{S}$ est ordonnançable, et nous donnons figure 7.13 une séquence d'ordonnancement (valide).

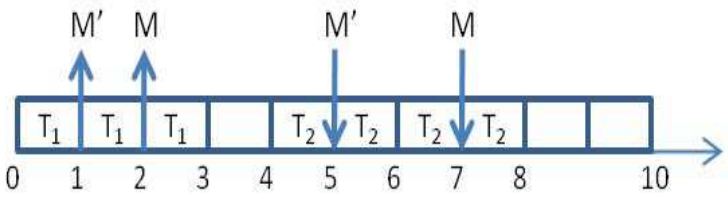


Figure 7.13 – Une séquence d'ordonnancement valide de $\overline{S}$

En utilisant l'approche arborescente qui prend en compte la sémantique des tests conditionnels, il y a dans tous les cas violation de l'échéance temporelle.

En effet, dans le cas où c'est le comportement $Comp_{11}$ ($Comp_{12}$) qui est choisi par la tâche T_1, le comportement $Comp_{21}$ ($Comp_{22}$) ne peut pas être choisi par T_2, car ils sont incompatibles du fait que $x \leq 100$ et $x > 100$ ($x > 100$ et $x \leq 100$).
Le message M (M') envoyé ne sera donc pas reçu sur la période d'étude. T_2 ne peut donc pas terminer son exécution.
Ainsi, une analyse du système en tenant compte des aspects arborescent et sémantique permet de conclure que le système n'est pas ordonnançable.

Nous concluons ce chapitre sur la comparaison des approches de validation en proposant :

1. si l'application ne comporte pas de tâche avec des instructions conditionnelles, alors utiliser l'approche de validation linéaire ;
2. sinon, utiliser l'approche de validation arborescente.

Bilan

Nous avons montré dans ce chapitre dans quelle mesure il pouvait être nécessaire d'utiliser notre approche de validation arborescente des applications temps-réel.
Rappelons le, cette approche consiste à produire un arbre d'ordonnancement de l'application à valider.

Dans le chapitre 6 où nous avons défini ces arbres, nous avons proposé une approche *brute*, exponentielle, et donc difficile à mettre en œuvre dans la pratique, d'obtention de ces arbres.
Dans le chapitre 8, nous allons nous intéresser à une méthode d'obtention basée modèle, et pour laquelle la complexité peut être réduite.

Chapitre 8

Mise en œuvre avec les réseaux de Petri

"*Science sans conscience n'est que ruine de l'âme*"
François Rabelais

Sommaire

Dans le chapitre 6, nous avons proposé une approche de génération *brute* et *directe* des arbres d'ordonnancement.
Cette approche est limitée :

1. elle génère tous les arbres sans soucis de corrections, et il faut ensuite faire le tri pour ne garder que ceux qui sont valides.
   Donc, on génère beaucoup trop d'arbres ;
2. elle est exponentielle en le nombre de tâches constituant l'application qu'on veut valider.
   Elle est donc impraticable ;
3. enfin, si l'objectif est d'obtenir des arbres *qualitatifs* (des arbres respectant certaines propriétés qualitatives comme par exemple le temps de réponse moyen de certaines tâches soit minimal), modifier le générateur d'arbres s'avère difficile.

L'idée est donc d'avoir un outil qui ne génère que des arbres valides, qui peut générer directement des arbres qualitatifs, et dont la complexité soit réduite.
Dans ce chapitre, nous proposons une approche de génération des arbres d'ordonnancement.
Cette approche utilise la puissance de modélisation des réseaux de Petri pour modéliser l'application et exploite les outils d'analyse associés, en particulier le graphe des marquages, pour analyser le comportement de l'application modélisée, et en extraire les arbres d'ordonnancement.
Ce chapitre se divise en trois parties :

1. nous commençons par présenter notre approche de modélisation des systèmes temps-réel en utilisant les réseaux de Petri ;
2. ensuite, nous prouvons que le modèle obtenu respecte bien les propriétés attendues ;
3. enfin, nous montrons comment en partant du pseudo-code des applications temps-réel, notre modèle est obtenu.

8.1 Modélisation par réseaux de Petri

Nous étendons les modèles de [Grolleau 1999] et [Pailler 2006] présentés au chapitre 3, afin de prendre en compte les relations intra-tâches et inter-tâches.
Nous rappelons que le modèle de réseau de Pétri que nous utilisons est caractérisé (voir définition 3.1.1) par le couple (N, M_0) où $N = (Q, T, W)$ $(Q \cap T = \emptyset)$ avec Q qui est l'ensemble fini de places, T l'ensemble fini de transitions,
$W : QXT \cup TXQ \Rightarrow \mathbb{N} \cup \{\mathbf{a}, \mathbf{b}, \mathbf{a}{+}\mathbf{b}\}$ la fonction de valuation et
$M_0 : Q \Rightarrow \mathbb{N} \cup \{\mathbf{a}, \mathbf{b}, \mathbf{a}{+}\mathbf{b}\}$ le marquage initial du réseau, où $\{\mathbf{a}{+}\mathbf{b}\}$ désigne l'ensemble $\{\mathbf{a}, \mathbf{b}\}$.
De plus, il est *P-coloré*, simplifié (nous utilisons juste deux couleurs **a** et **b**), avec ensemble terminal, et fonctionne sous la règle du tir maximal.
La règle du tir maximal permet de modéliser le temps, le marquage initial du réseau permet en particulier de prendre en compte les dates de premier réveil, et l'ensemble terminal permet de gérer les contraintes liées aux échéances.
Contrairement aux réseaux de [Grolleau 1999] et [Pailler 2006] qui comportent deux (2) couches, notre réseau de Petri comporte trois (3) parties :

1. une couche structurelle pour la représentation du système de tâches ;
2. une couche temporelle pour la modélisation de la dynamique du système ;
3. une couche sémantique pour la modélisation des relations d'incompatibilité entre les différents comportements des différentes tâches, et pour assurer la cohérence sémantique de l'application.

8.1.1 Modélisation de la couche structurelle

Nous avons envisagé deux approches :

1. une approche chemin qui consiste à considérer une tâche comme un ensemble de chemins.
   Il s'agit en fait de l'ensemble de ses comportements (effectifs).
   Cette approche renvoie à la notion de comportements du chapitre 5. C'est sur cette approche que notre modèle a été validé.
   Dans ce cas, la relation d'incompatibilité implémentée est la relation $\mathcal{RI}$.
   Son avantage est sa facilité de manipulation et d'implémentation.
   Cette approche a par contre un inconvénient : elle est redondante, parce qu'elle duplique les blocs indépendants (qui ne comportent pas de conditionnelles)

dans chacun des comportements.
C'est la raison pour laquelle nous avons envisagé l'étude de l'approche arbre ;

2. l'approche arbre est celle utilisée par [Pailler 2006].
Dans ce travail, nous la complétons pour qu'elle prenne en compte la sémantique des tests.
Cette approche renvoie à la notion d'arbre d'exécution du chapitre 5.
Dans ce cas, la relation d'incompatibilité implémentée est la relation $\mathcal{RI}_{min}$.
Les tâches sont ici modélisées par leur arbre d'exécution, ce qui limite le nombre de places et de transitions dans la couche structurelle, par rapport à l'approche chemin.

8.1.1.1 Approche chemin

Dans cette approche, chaque tâche T_i de l'application est constituée d'un ensemble de chemins effectifs (nous rappelons que nous les appelons aussi comportements effectifs) $Compef f_i$.
Nous donnons figure 8.3 un exemple de modélisation suivant l'approche chemin du système de deux tâches de la figure 8.1 dont la représentation arborescente est donnée figure 8.2.

```
Tâche T1
b(1);
S(M1);
If (x<5) [1] then
        b(1);
Else
        L(R1);
        b(1);
        U(R1);
        if (y>2) [1] then
                b(2);
        else
                b(1);
        endif;
Endif;
End;
```

```
Tâche T2
R(M1);
If (z<5) [1] then
        b(2);
Else
        b(3);
Endif;
End;
```

Figure 8.1 – Pseudo-code d'un système de 2 tâches : T_1 a 3 comportements et T_2 en a 2, de plus, T_1 utilise la ressource R et les 2 tâches échangent le message M

Nous pouvons remarquer que chaque comportement consiste en une succession finie de places et de transitions, qui correspondent aux durées des blocs généraux rencontrés.
La modélisation de ces blocs (et donc de leur durée) se fait suivant la technique présentée figure 3.6.
Nous rappelons qu'elle consiste à modéliser un bloc de durée 1 par une place et une transition, un bloc de durée 2 par deux places et deux transitions et un bloc de durée plus grande ou égale à 3 par trois places et trois transitions.

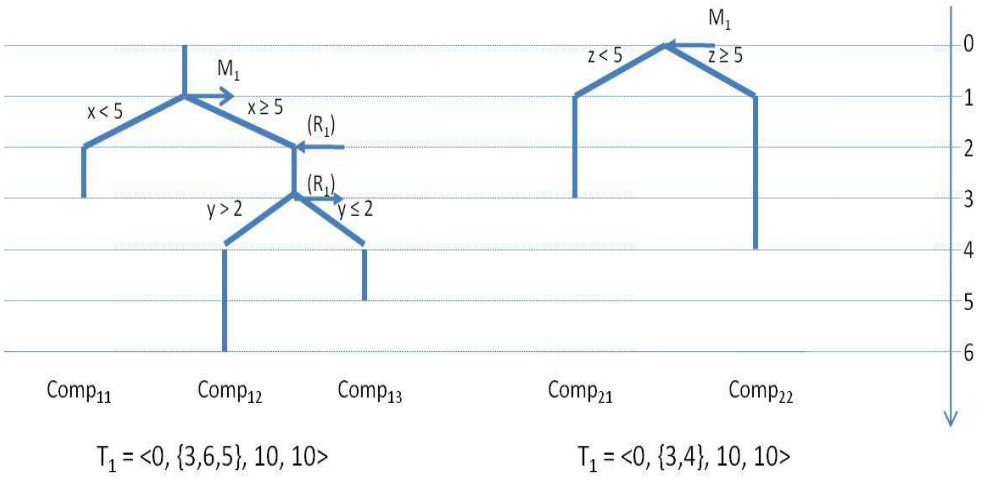


Figure 8.2 – Représentation arborescente du système de tâches de la figure 8.1

La modélisation des tests conditionnels (et donc de leur durée), se fait en les considérant comme des blocs indépendants.
Rappelons toutefois que pour les tests de durée plus grande que 1, nous utilisons au préalable les conventions adoptées section 4.1.1, afin de réduire la durée de chaque test à 1.
Enfin, les primitives temps-réel sont modélisées comme dans les approches de [Grolleau 1999] et [Pailler 2006].

Nous allons maintenant nous intéresser à une description plus formelle du modèle de réseau de Petri, approche chemin, utilisé pour modéliser nos applications temps-réel.
Nous allons donc par la suite spécifier les éléments des ensembles (N, M_0).
Soit $i \in 1, \ldots, n$, où n est le nombre de tâches de l'application que nous voulons modéliser.
Tous les comportements de la tâche T_i partagent la place $Activ_i$ qui les met en exclusion mutuelle, car deux comportements d'une tâche ne peuvent être suivis simultanément.

L'ensemble des places est
$Q_{P_i} = \{Activ_i, P_{ijkl}\ tel\ que\ k \in 1, \ldots, 3,\ j \leq C_{il},\ l \in 1, \ldots, |Compeff_i|\}$ où :

1. C_{il} est la durée du comportement $Comp_{il}$;
2. $|Compeff_i|$ est le nombre de comportements (effectifs) de la tâche T_i ;
3. $Activ_i$ gère l'activation de la tâche T_i et la non ré-entrance.
   La non ré-entrance est évitée du fait qu'il faut un jeton de couleur **b**, qui signifie que l'instance précédente est terminée, avant de démarrer une nouvelle instance ;
4. P_{ijkl} désigne la j^{eme} place de la tâche T_i se trouvant sur le chemin l.

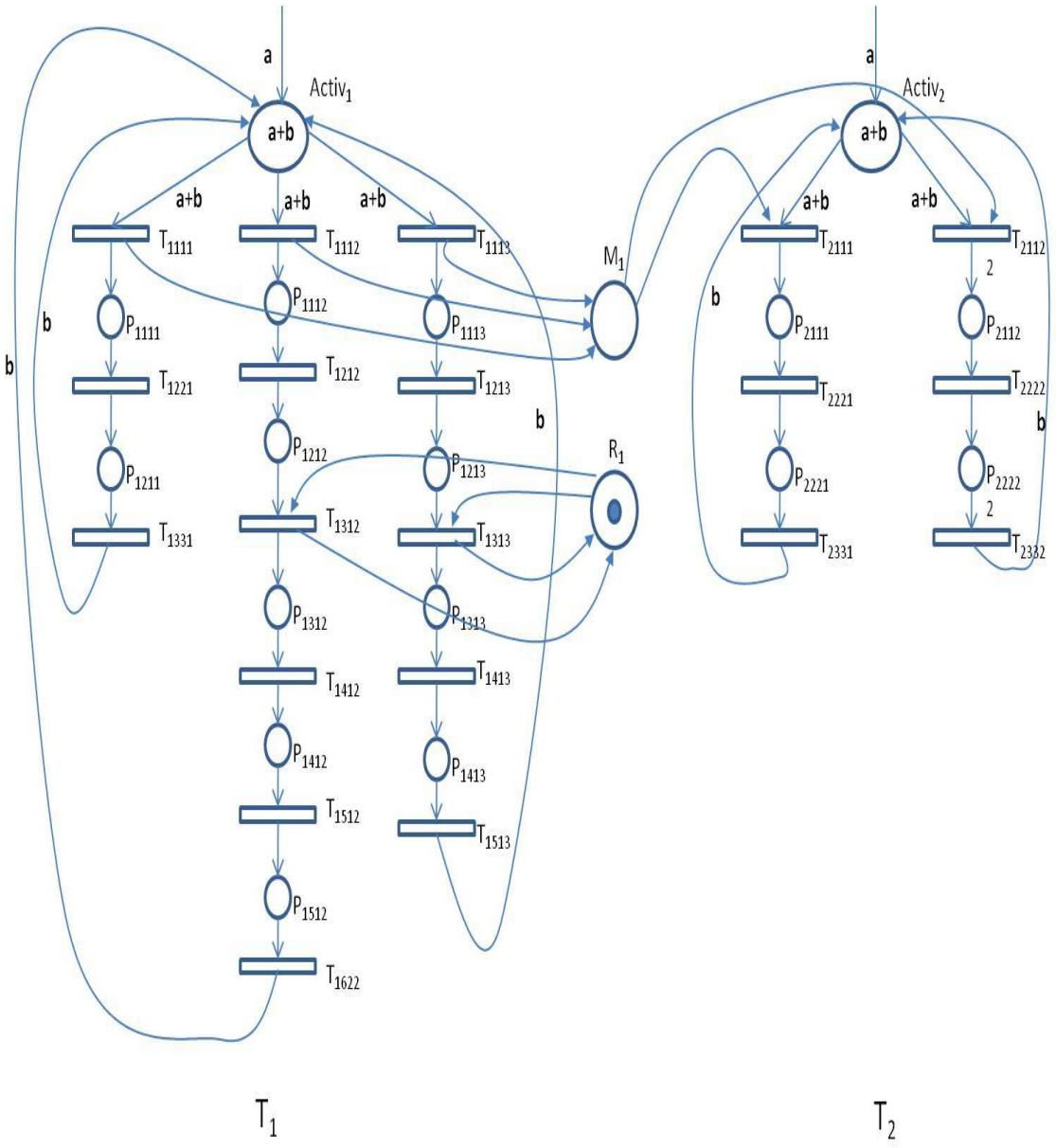


Figure 8.3 – Modélisation par réseau de Petri, approche chemin, du système de tâches de la figure 8.1. Pour des raisons de lisibilité, la place *Processor* n'est pas représentée

k est utilisé pour gérer la compression des places, afin d'éviter une explosion en nombre de places de la modélisation (voir figure 8.4).

Nous voulons aussi modéliser l'échange de messages et le partage de ressources entre les tâches de l'application.
Nous complétons pour cela l'ensemble Q_{P_i} par l'ensemble Q_I des places nécessaires pour prendre en compte ces interactions.
$Q_I = \{Processor,\ R_r,\ , M_m\ tel\ que\ r \in 1 \ldots |\mathcal{R}|,\ m \in 1 \ldots |\mathcal{EM}|\}$ où :

1. la place $Processor$ permet de modéliser le processeur ;
2. chaque place R_r modélise une ressource.
   Il y a autant de places R_r que de ressources utilisées dans l'application ;
3. chaque place M_m correspond à un message, donc à une boite à lettres.
   Il y a autant de places M_m que de messages envoyés.

L'ensemble des transitions est
$T_{P_i} = \{T_{ijkl}\ tel\ que\ k \in 1 \ldots 3,\ j \leq C_{il},\ l \in 1, \ldots, Compeff_i|\}$ où :

1. T_{ijkl} désigne la j^{eme} transition de la tâche T_i se trouvant sur le chemin l.
   k est défini, comme pour les places, par le schéma de la figure 8.4.

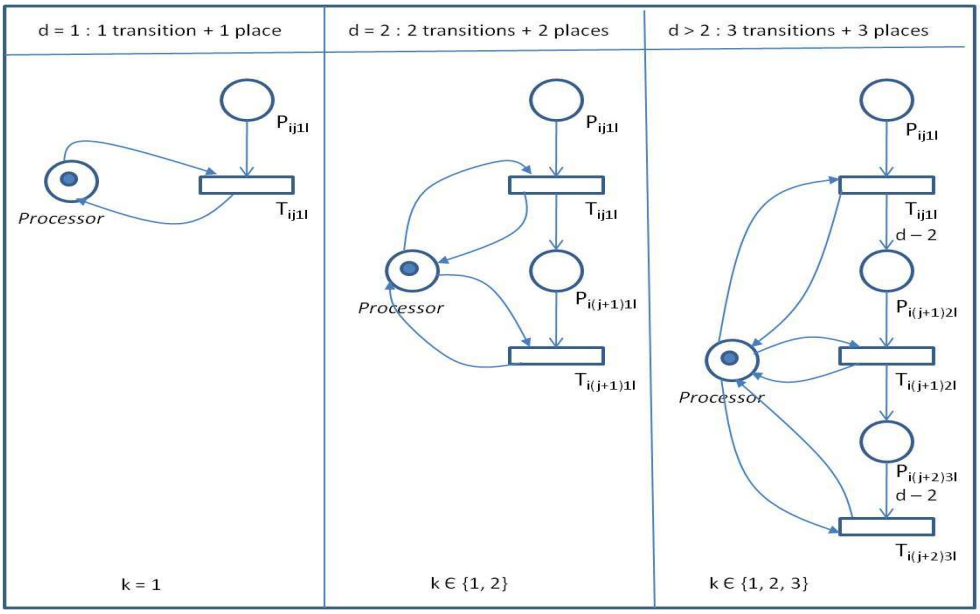


Figure 8.4 – Prise en compte des blocs de durée d. Le coefficient k varie de 1 à 3 en fonction de la longueur du bloc à modéliser

La fonction de valuation W est définie sur cette partie par :

1. $W(Activ_i, T_{i11l}) = \mathbf{a}+\mathbf{b}$, où, $\mathbf{a}+\mathbf{b}$ désigne l'ensemble $\{\mathbf{a}, \mathbf{b}\}$.
   Cette égalité traduit le fait qu'une activation n'est possible qu'après le dépôt des jetons **a** (la tâche a été réactivée) et **b** (l'instance précédente est terminée) dans la place $Activ_i$;

2. $W(Fin, Activ_i) = \mathbf{b}$, pour chaque transition Fin[1] des différents chemins (pour marquer la fin de l'exécution de l'instance en cours) ;
3. $W(T_{ijkl}, Processor) = W(Processor, T_{ijkl}) = 1$.
   Nous modélisons ainsi le fait que le processeur est occupé à chaque instant par une seule tâche ;
4. $W(R_r, T_{ij1l}) = 1$ et $W(T_{ij'k'l}, R_r) = 1$.
   Ceci traduit le fait que toute ressource prise en début d'un bloc j (remarquons que $k = 1$ pour signifier que les ressources ne peuvent pas être prises à l'intérieur des blocs *compressés*), est libérée à la fin d'un bloc j' du même comportement $Comp_{il}$ de la tâche T_i.
   Remarquons que nous pouvons généraliser ce résultat avec des valuations supérieures à 1, pour signifier que l'on prend (libère) plusieurs instances de la ressource ;
5. pour toute tâche T_i qui envoie un message M_m sur le chemin $Comp_{il}$ à une tâche $T_{i'}$ sur le chemin $Comp_{i'l'}$, nous avons $W(T_{ijkl}, M_m) = 1$ si le message est envoyé à la fin du bloc j de la tâche T_i, et $W(M_m, T_{i'j'1l'}) = 1$ si un message est reçu au début du bloc j' de la tâche $T_{i'}$.

Le marquage initial des places est :

1. $M_0(Activ_i) = \mathbf{a}+\mathbf{b}$ si $r_i = 0$ et $M_0(Activ_i) = \mathbf{b}$ si $r_i > 0$.
   Nous rappelons que dans le cadre de cette thèse, $r_i \leq D_i \leq P_i$ (le cas $r_i > P_i$ est traité dans [Grolleau 1999]).
   Les marquages initiaux de la place $Activ_i$ s'expliquent :
   (a) si $r_i = 0$, la tâche T_i est active dès le début.
   Sa place d'activation doit contenir un jeton **a** et un jeton **b** pour qu'elle puisse s'exécuter.
   D'où $M_0(Activ_i) = \mathbf{a}+\mathbf{b}$;
   (b) si $r_i > 0$, la tâche T_i est à départ différé et doit être retardée.
   Elle contient donc initialement le jeton **b** (car il n'y a pas d'instance en cours) et attend le jeton **a** afin d'être active.
   Donc $M_0(Activ_i) = \mathbf{b}$;
2. $M_0(P_{ijkl}) = 0$, pour toute place P_{ijkl} ;
3. $M_0(Processor) = 1$, car nous sommes dans l'hypothèse monoprocesseur ;
4. $M_0(R_r) = |R_r|$, où $|R_r|$ désigne le nombre d'instances de la ressource R_r ;
5. $M_0(M_m) = 0$, car les boites à lettres sont initialement vides.

8.1.1.2 Approche arbre

Dans cette approche, le réseau de Petri utilisé pour représenter le système de tâches suit le modèle arborescent de la tâche.

[1] Le terme *Fin* est utilisé par souci de simplification, et désigne la dernière transition de chaque chemin

Nous donnons figure 8.5 le réseau de Petri, approche arbre, représentant le système de tâches de la figure 8.1, où t_1 désigne $(x < 5)_{T_1}$, t_2 désigne $(y > 2)_{T_1}$ et t_3 désigne $(z < 5)_{T_2}$.

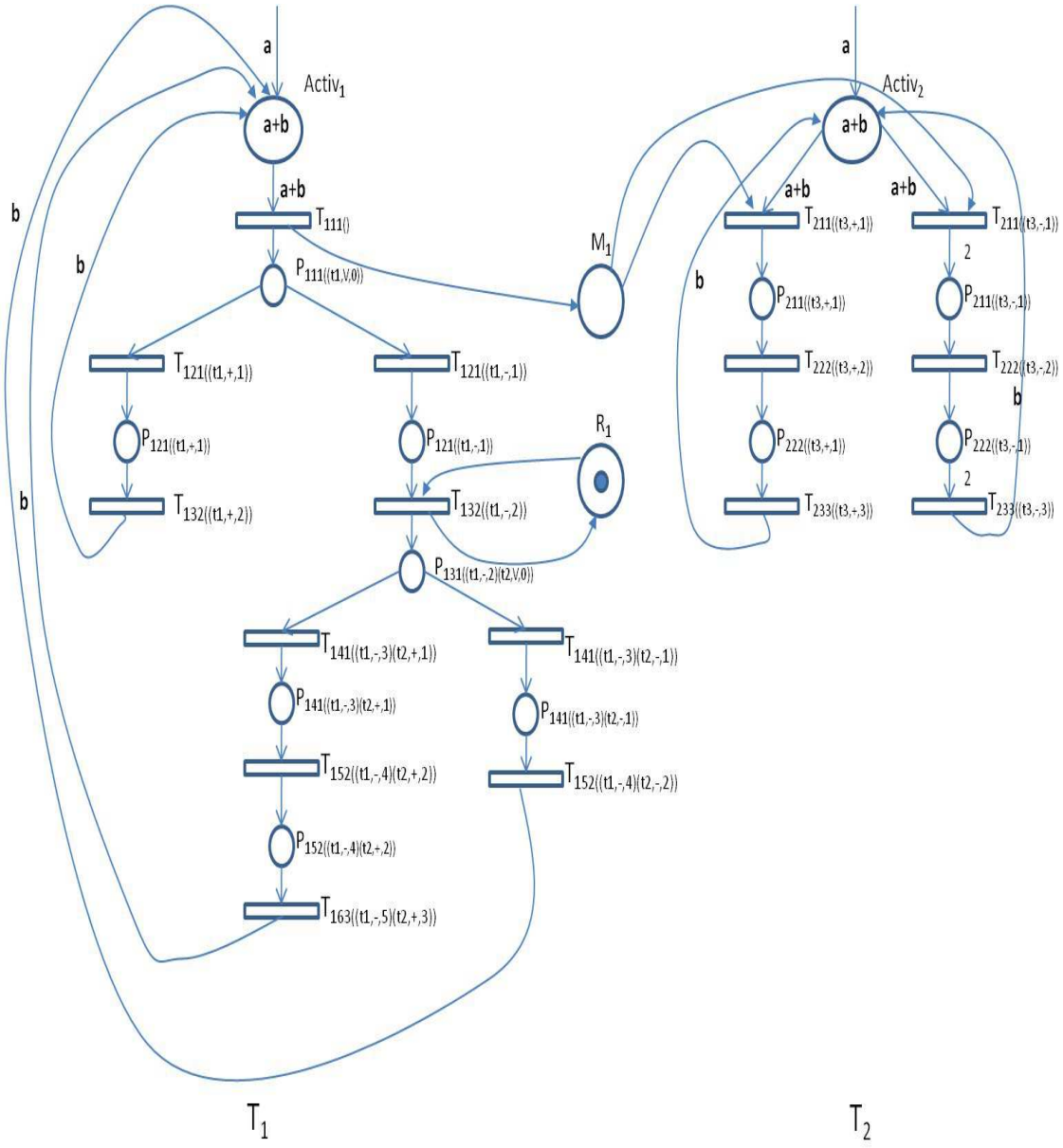


Figure 8.5 – Modélisation par réseau de Petri, approche arbre, du système de tâches de la figure 8.1. Pour des raisons de lisibilité, la place *Processor* n'est pas représentée

Nous allons décrire ce réseau de façon formelle, et en le simplifiant, pour ne pas répéter les points similaires à l'approche chemin.

Nous commençons par remarquer que, par analogie au modèle d'exécution des tâches, approche arbre (voir section 4.2.3), un nœud correspond à une place et un lien correspond à une transition.

L'ensemble des places est
$Q_{P_i} = \{Activ_i, P_{ijkL}\ tel\ que\ k \in 1, \ldots, 3,\ j \leq C_{il},\ l \in 1, \ldots, |Compeff_i|\}$ où :

1. i désigne le numéro de la tâche ;
2. j désigne le numéro de la place sur le comportement induit par L ;
3. k est utilisé pour la compression des places comme indiqué figure 8.4 ;
4. autant que dans l'approche chemin, nous avons identifié pour chaque place le comportement sur lequel elle se trouvait (en utilisant l), nous avons besoin d'identifier dans l'approche arbre les branches sur lesquelles se trouvent les places (nous utilisons L).
   L est une liste de triplets $(test, signe, position)$ où :
   (a) *test* correspond au test qui engendre la branche qui comporte la place P_{ijkL} ;
   (b) *signe* spécifie le comportement sur lequel se trouve la place :
      i. si $signe = +$, alors P_{ijkL} se trouve sur la branche induite par le résultat positif du test *test* ;
      ii. si $signe = -$, alors P_{ijkL} se trouve sur la branche induite par le résultat négatif du test *test* ;
      iii. si $signe = V$ (pour vide), alors P_{ijkL} représente la place occupée par le test *test* ;
   (c) *position* correspond au nombre de places rencontrées depuis le test *test*, sur le comportement induit par *test* et *signe*.

Si $L = ()$ (L est vide), cela veut dire qu'aucun test n'a encore été rencontré.
Si tous les tests de la tâche T_i sont de $Niveau = 1$ (voir définition 7.3.1), L aura à chaque instant 1 seul élément.
Sinon, s'il en existe de $Niveau = 2$, alors, L aura à un moment maximum 2 éléments, et ainsi de suite.

L'ensemble des transitions est
$T_{P_i} = \{T_{ijkL}\ tel\ que\ k \in 1 \ldots 3,\ j \leq C_{il},\ l \in 1, \ldots, Compeff_i|\}$ où :

1. i désigne le numéro de la tâche ;
2. j désigne le numéro de la transition sur le comportement induit par L ;
3. k est utilisé pour la compression des transitions comme indiqué figure 8.4 ;
4. comme pour l'ensemble des places, L est utilisé ici pour repérer les transitions liées aux tests.
   L est une liste de triplets $(test, signe, position)$ où :

(a) *test* correspond au test qui engendre la branche qui comporte la transition T_{ijkL} ;

(b) *signe* spécifie le comportement sur lequel se trouve la transition :

i. si $signe = +$, alors T_{ijkL} se trouve sur la branche induite par le résultat positif du test *test* ;

ii. si $signe = -$, alors T_{ijkL} se trouve sur la branche induite par le résultat négatif du test *test*.

Dans ce cas, *signe* ne peut pas être vide ;

(c) *position* correspond au nombre de transitions rencontrées depuis le test *test*, sur le comportement induit par *test* et *signe*.

Si $L = ()$ (L est vide), cela veut dire qu'aucun test n'a encore été rencontré.
Si tous les tests de la tâche T_i sont de $Niveau = 1$ (voir définition 7.3.1), L aura à chaque instant un seul élément.
Sinon, s'il en existe de $Niveau = 2$, alors, L aura à un moment maximum deux éléments, et ainsi de suite.

Le marquage initial des places et la **fonction de valuation** se définissent de la même manière que dans l'approche chemin.

Nous pouvons remarquer que dans l'approche arbre, le réseau comporte moins de places et de transitions que dans l'approche chemin.
Cette différence s'explique par le fait que, les aspects *redondants*, dus aux branches indépendantes qui se répétaient dans les différents comportements dans l'approche chemin, n'existent pas dans l'approche arbre.

8.1.1.3 Modélisation de la tâche oisive

La modélisation complète de la couche structurelle nécessite de prendre en compte la tâche oisive (voir section 5.4).
Nous rappelons qu'il s'agit d'une tâche nous permettant de modéliser l'inactivité processeur.
Nous avons retenu la méthode de la plus courte durée (voir section 5.4.2) : elle consiste à construire la tâche oisive en choisissant dans chaque tâche le comportement qui a la plus longue durée d'exécution (cela revient à considérer le facteur de charge maximal U_{max} de l'application[2]).
Nous distinguons deux cas :

1. si $P(1 - U_{max}) \geq 0$,
   la structure de la tâche oisive est modélisée par une place ($Activ_0$) et une transition ($Oisif_0$) (voir figure 8.6) :

[2]Par opposition au facteur de charge U_{min} qui correspond au choix de la plus courte branche dans chaque tâche de l'application

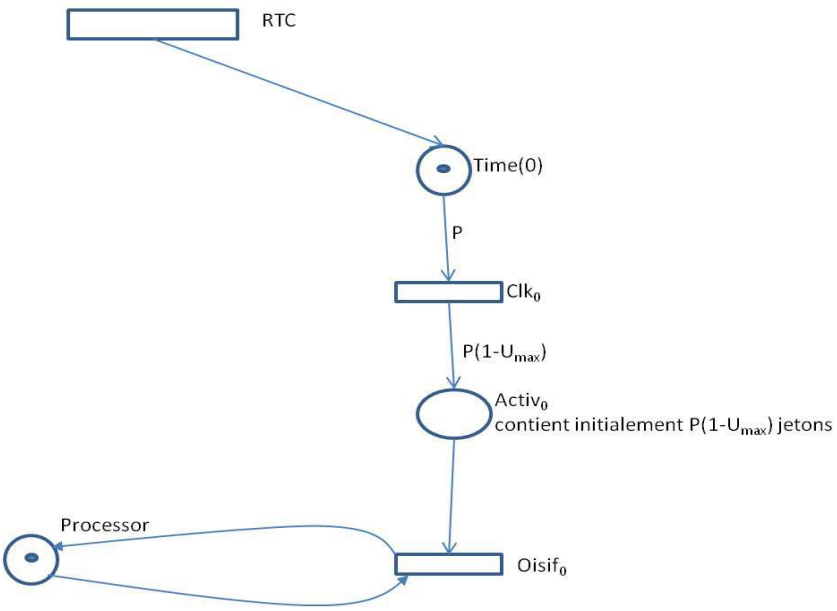


Figure 8.6 – Prise en compte de la tâche oisive dans le cas où $P(1 - U_{max}) \geq 0$

(a) la place $Activ_0$ contient initialement $P(1 - U_{max})$ jetons.
Nous savons qu'il y aura en effet $P(1 - U_{max})$ temps d'inactivité processeur sur l'hyperpériode.
Dans le cas où la branche choisie pendant l'exécution de la tâche n'est pas celle induisant la plus longue durée, nous allongeons la durée de la tâche oisive, afin d'avoir la durée maximale, en rajoutant des jetons dans la place $Activ_0$.
Cela permet l'exécution de toute la tâche oisive initiale, en prenant en compte les fluctuations de durées dues au choix des branches conditionnelles ;

(b) la transition $Oisif_0$ est liée à la place $Processor$ (comme toutes les transitions de la couche structurelle).
Son tir caractérise l'exécution de la tâche oisive ;

2. sinon, on a $P(1 - U_{max}) < 0$:

(a) si en choisissant pour chaque tâche n'importe quel comportement, le facteur de charge résultant U_{res} permet d'avoir $P(1 - U_{res}) \leq 0$, alors la tâche oisive n'a pas lieu d'exister, et elle ne sera pas représentée ;

(b) sinon, il existe au moins une combinaison de comportements qui permet d'obtenir $P(1 - U_{res}) > 0$, la tâche oisive est modélisée en utilisant l'approche présentée figure 8.7.
Ainsi, la tâche oisive est modélisée suivant l'approche présentée figure 8.7 si $U_{min} \leq 1$ et $U_{max} > 1$.
La structure de la tâche oisive dans ce cas est modélisée par trois places

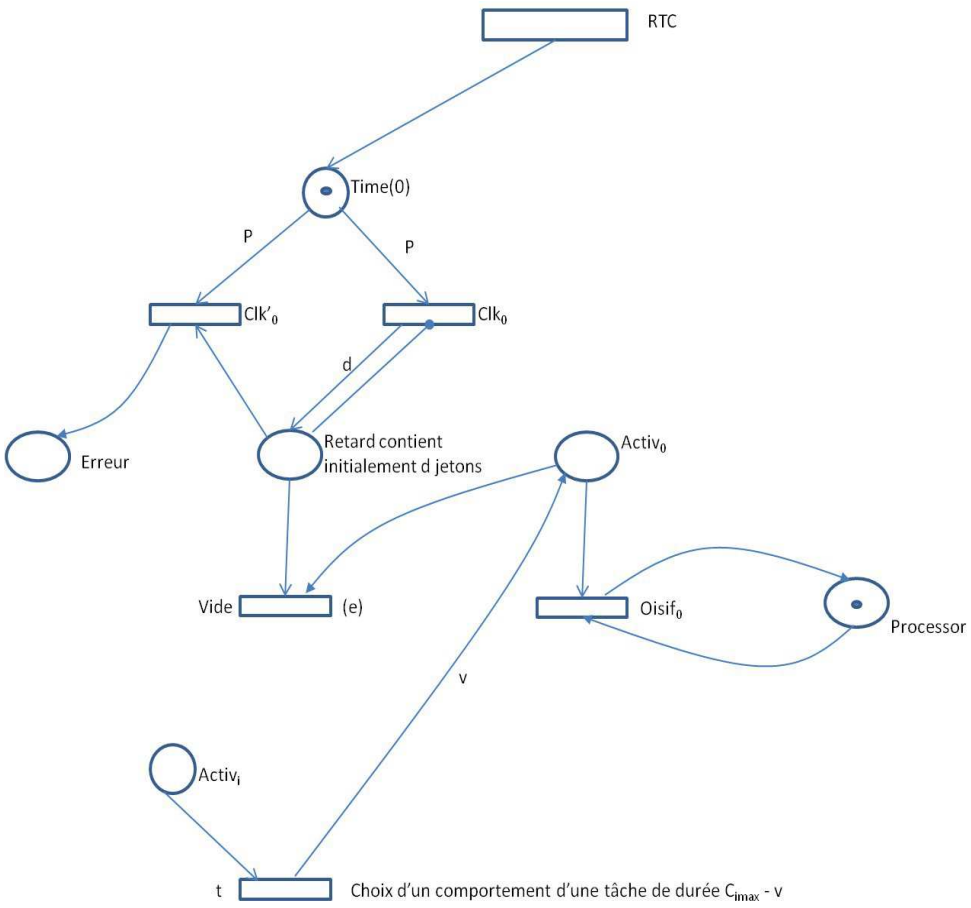


Figure 8.7 – Prise en compte de la tâche oisive dans le cas où $U_{max} > 1$ et $U_{min} \leq 1$ et $d = |P(1 - U_{max})|$

($Activ_0$, $Retard$ et $Erreur$) et deux transitions ($Oisiv_0$ et $Vide$).
Comme $P(1 - U_{max}) < 0$, Clk_0 ne dépose plus de jetons initialement dans la place $Activ_0$.
Il y a donc un retard d'exécution qu'il faut prendre en compte.
Pour cette raison, nous créons une place $Retard$ qui contient initialement $d = |P(1 - U_{max})|$ jetons.
La transition Clk_0 y dépose d jetons chaque fois qu'elle est tirée.
Si pendant l'exécution, un comportement d'une tâche de durée $C_{imax} - v$ est choisi (comportement dont la première transition est notée t sur notre figure), nous déposons v jetons dans la place $Activ_0$.
Si ces v jetons ne sont pas tous consommés par le tir de la transition $Oisiv_0$, nous créons une transition $Vide$, synchronisée sur l'évènement e qui est un évènement toujours occurrent.
Le tir de cette transition réinitialise la place $Activ_0$.
Cette synchronisation de l'évènement e sur la transition Vide implique que la transition Vide, sitôt validée, est aussitôt tirée.
On tire donc en fait t suivi de Vide $Min(d, v)$ *fois.*
La place $Retard$ devra être vide lors du prochain tir de Clk_0, ce que modélise l'arc inhibiteur.
La transition Clk_0 ne pourra donc être tirée que quand la place $Retard$ n'aura plus de jetons, et son tir déposera d jetons dans la place $Retard$.
Sinon, cela signifiera que pour l'hyperpériode précédente, on aura eu $U_{eff} > 1$ (le facteur de charge effectif plus grand que 1). Il y aura donc eu une erreur.
Pour la modéliser, nous complétons le modèle de la tâche oisive en lui ajoutant une transition Clk'_0 et une place $Erreur$ dont le marquage initial est égal à 0 ($M_0(Erreur) = 0$).
Si $M(Erreur) > 0$, cela signifie qu'une erreur temporelle a eu lieu.
Le tir de Clk'_0 signifie que l'application n'est pas ordonnançable (le facteur de charge de l'ensemble des comportements choisis par les tâches est plus grand que 1).

Remarque 8.1.1 Si on considère le fonctionnement du réseau de Petri avec marquage terminal, on n'arrivera jamais à ce stade, puisqu'il aura fallu préalablement passer par des marquages violant l'ensemble terminal.

Nous présentons et expliquons sur deux exemples (exemples 8.1.1.1 et 8.1.1.2) le principe de modélisation de la tâche oisive.
L'exemple 8.1.1.1 s'applique aux tâches modélisées en utilisant l'approche chemin, et l'exemple 8.1.1.2 s'applique aux tâches modélisées en utilisant l'approche arbre.

Exemple 8.1.1.1 Tâche oisive, approche chemin
Nous considérons le système de tâches de la figure 8.1.
La tâche oisive a pour paramètres temporels $T_0 = < 0, 0, 10, 10 >$, car
$P(1 - U_{max}) = 10(1 - (\frac{6}{10} + \frac{4}{10})) = 0$.

Par conséquent, la tâche oisive est représentée par la place $Activ_0$ et $Oisif_0$ comme représentée figure 8.8.

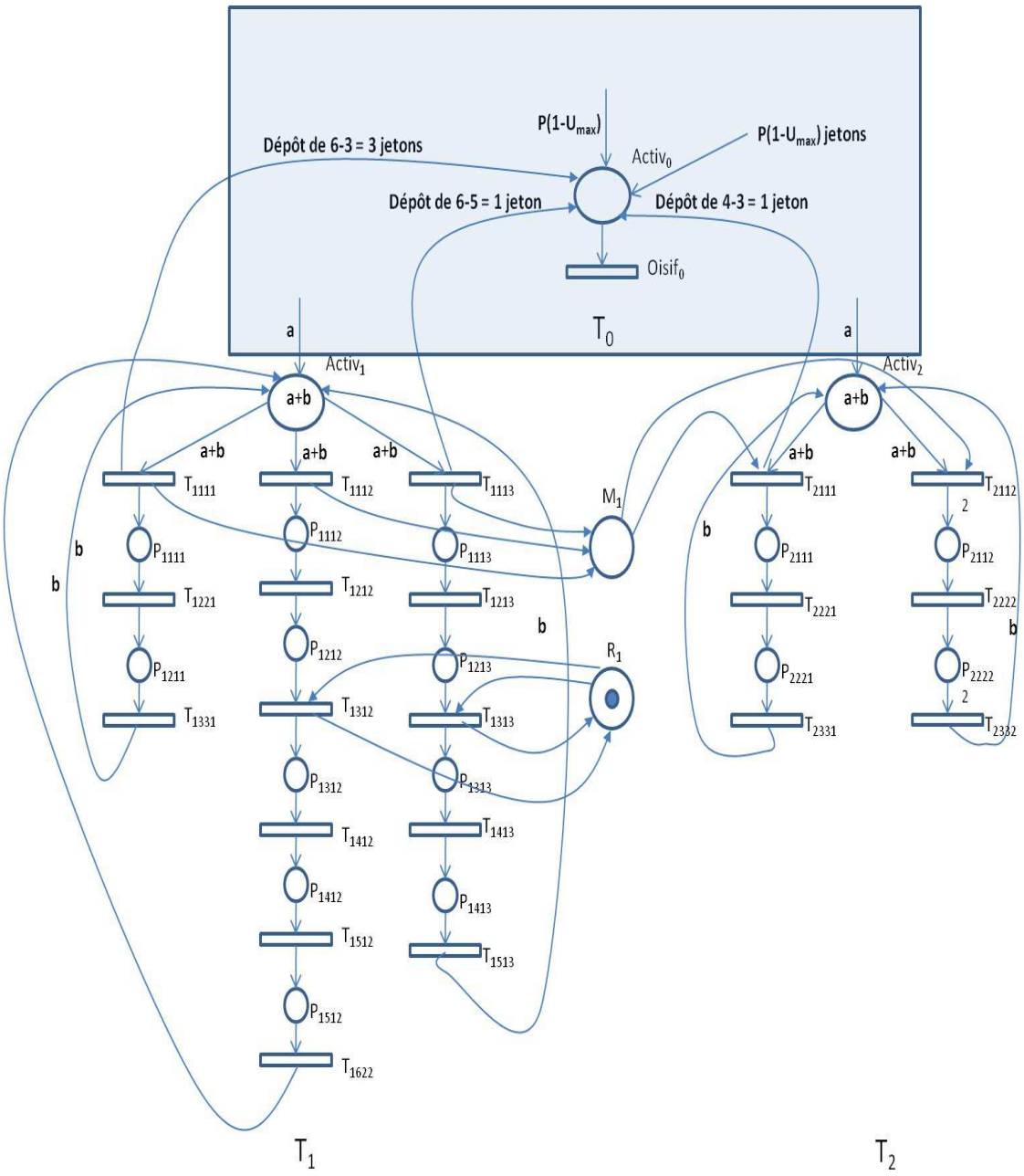


Figure 8.8 – Prise en compte de la tâche oisive du système de tâches de la figure 8.1 suivant l'approche chemin (cas où $P(1 - U_{max}) \geq 0$). Pour des raisons de lisibilité, la place *Processor* n'est pas représentée

De façon générale, nous connaissons $C_{il_{max}}$ qui correspond à la durée d'exécution

du comportement le plus long de la tâche T_i.
L'objectif est d'assurer que quelque soit le comportement choisi, l'ensemble de tous les temps creux est bien pris en compte sur toute l'hyperpériode.
Si la durée d'un comportement choisi est strictement plus petite que la durée du comportement le plus long, il faut donc permettre la complétion des durées restantes sur l'hyperpériode en rajoutant les temps creux manquants.
Cela se traduit de façon formelle par :
$\forall\ i \in 1, \ldots, n, \forall\ l \in 1, \ldots, |Compef f_i|$, si $C_{il} < C_{il_{max}}$, alors
$W(T_{111l}, Activ_0) = C_{il_{max}} - C_{il}$.

Exemple 8.1.1.2 Tâche oisive, approche arbre
Nous considérons de nouveau le système de tâches de la figure 8.1, et donnons figure 8.9 notre représentation de la tâche oisive, quand l'approche de modélisation basée arbre est considérée.
Nous devons ici prendre en compte un autre critère : le choix d'un comportement ne se fait pas en début de tâche, comme dans l'approche chemin.
Nous raisonnons de façon différente.
Il ne s'agit plus de considérer la durée d'exécution du comportement le plus long, mais de tenir compte de la durée d'exécution de la plus longue branche chaque fois que nous exécutons un test.
À chaque embranchement, nous rajoutons à la place $Activ_0$ la différence entre la durée d'exécution si on avait choisi la plus longue branche et la durée d'exécution de la plus longue branche de l'embranchement choisi.
Dans le cas où l'embranchement choisi n'a qu'une seule branche, la plus longue branche est cette unique branche.
Cela permet, dès qu'une branche est choisie, de compléter les temps restants sur la longueur de l'hyperpériode par des temps creux.
Ainsi, pour la figure 8.9, au niveau du test t_1, la différence entre la plus longue branche si on choisissait la branche *négative* et la durée d'exécution de la branche choisie vaut $6 - 3 = 3$. Nous rajoutons donc 3 jetons à la place $Activ_0$.
Nous appliquons le même principe au niveau des tests t_2 et t_3.

Les exemples 8.1.1.1 et 8.1.1.2 n'illustrent pas le cas où $P(1 - U_{max}) < 0$.
Ce cas sera traité dans les figures 8.14 et 8.15.

Nous concluons cette section sur la modélisation des tâches oisives en énonçant le résultat 8.1.1 concernant la modélisation de la tâche oisive dans le cas de tâches à départs différés (temps creux acycliques).

Résultat 8.1.1 [Pailler 2006] Prise en compte des tâches à départs différés
La prise en compte des tâches conditionnelles à départ différés se fait de la même façon que lorsqu'on considère le pire cas.
Cela veut dire que la modélisation des temps creux acycliques ne change pas, selon que l'on soit dans l'approche pire cas (celle qui consiste à considérer en présence d'un bloc conditionnel sa représentation linéaire), ou que l'on considère explicitement les instructions conditionnelles (approche arborescente).

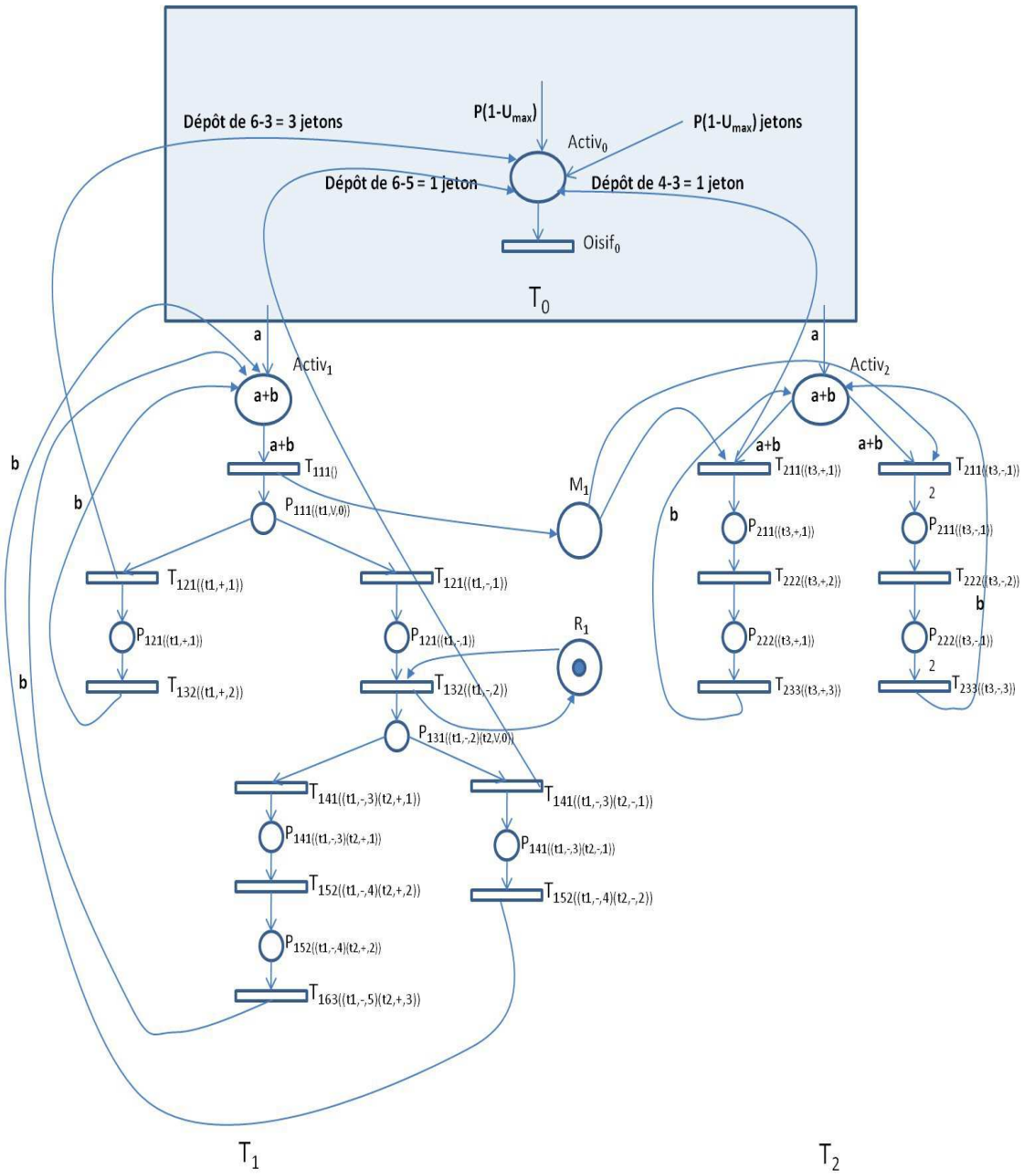


Figure 8.9 – Prise en compte de la tâche oisive du système de tâches de la figure 8.1 suivant l'approche arbre (cas où $P(1-U_{max}) \geq 0$). Pour des raisons de lisibilité, la place *Processor* n'est pas représentée

Ce résultat sera prouvé dans une prochaine étude, puisqu'il n'a pas été complètement démontré dans [Pailler 2006].

8.1.2 Modélisation de la couche temporelle

Nous rappelons que la couche temporelle permet de modéliser la gestion du temps.
Nous ajoutons aux places et transitions de la couche structurelle des places et des transitions afin de modéliser l'écoulement du temps.
Nous présentons de façon formelle le modèle initial de [Grolleau 1999] donné figure 8.10 pour prendre en compte le temps.
Ce modèle sera complété par la suite pour prendre en compte la sémantique des instructions conditionnelles.

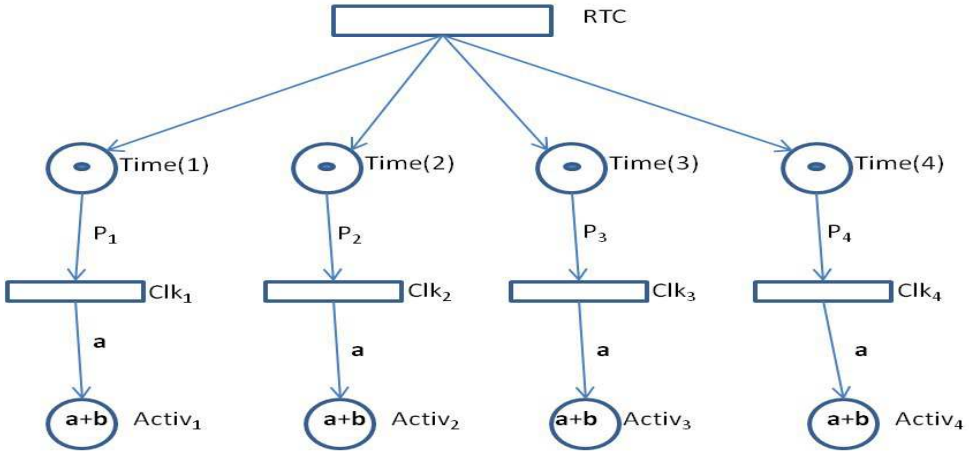


Figure 8.10 – Modélisation d'une couche temporelle pour un système de 4 tâches : $(Time(i), Clk_i)$ modélise l'horloge locale de la tâche T_i, et RTC représente l'horloge (globale) du système

L'ensemble des transitions ajoutées est
$T_T = \{RTC,\ Clk_i,\ i \in 1, \ldots, n\}$ où :

1. RTC est l'horloge globale du système.
   Grâce à la règle du tir maximal, chaque ensemble de transitions tiré contient RTC.
   RTC produit à chaque tir un jeton dans la place $Time(i)$. Ceci permet de disposer d'une modélisation du temps : chaque tir de RTC correspond à une unité de temps ;
2. Clk_i est utilisée pour réactiver la tâche à chaque période P_i, en déposant un jeton d'activation (**a**) dans la place $Activ_i$.

L'ensemble des places mises en œuvre dans la gestion de la périodicité des tâches est

$Q_T = \{Time(i),\ i \in 1, \ldots, n\}$ où :

1. chaque place $Time(i)$ sert de compteur local, et mémorise le temps écoulé depuis la dernière activation.
   Chaque tir de RTC dépose un jeton dans $Time(i)$, et toutes les P_i unités de temps, le tir de Clk_i est déclenché.

Nous complétons la **fonction de valuation** W en ajoutant de nouvelles règles :

1. $W(Time(i), Clk_i) = P_i$ (pour que les tâches soient activées périodiquement) ;
2. $W(RTC, Time(i)) = 1$.

Enfin, nous ajoutons au **marquage initial** :

1. $M_0(Time(i)) = P_i - r_i + 1$ si $0 < r_i \leq P_i$ et $M_0(Time(i)) = 1$ si $r_i = 0$.
   Cette propriété permet de gérer les dates de réveil des tâches qui peuvent être simultanées ou différées :
   (a) si $r_i = 0$, la tâche T_i doit être active dès le début de l'application d'où $M_0(Time(i)) = 1$;
   (b) sinon, elle doit être retardée de r_i unitées de temps avant d'être activée, et cela à chaque réactivation.
   D'où $M_0(Time(i)) = P_i - r_i + 1$.

La gestion du temps inclut aussi la prise en compte des **délais critiques**.
Afin de les modéliser, nous définissons un **ensemble terminal** $\mathcal{I}$.
Les propriétés de cet ensemble doivent être vérifiées par tous les marquages du réseau de Petri.

$$M \in \mathcal{I} \Leftrightarrow \begin{cases} 1.\ M(Time(i)) > D_i \Rightarrow M(Activ_i) = \mathbf{b}; \\ 2.\ M(Time(i)) = 1 \Rightarrow M(Activ_i) = \mathbf{a} + \mathbf{b}\ ou\ M(Activ_i) = \mathbf{b}; \\ 3.\ M(Time(i)) \leq P_i. \end{cases}$$

Les points 1 et 2 ont déjà été commentés section 3.3.2.
Nous avons ajouté le point 3 pour garantir la réactivation périodique des tâches.
La place $Time(i)$ ne doit pas contenir à un instant plus de P_i jetons.
Cette condition est rendue nécessaire d'une part par la gestion de la tâche oisive, et d'autre part par la gestion de la couche sémantique, comme nous le montrons dans le paragraphe 8.1.3.

Nous complétons la modélisation de la couche temporelle en prenant en compte les aspects temporels de la tâche oisive :

1. si $P(1 - U_{max}) \geq 0$,
   la transition Clk_0 dépose toutes les P (P est l'hyperpériode) unités de temps $P(1 - U_{max})$ jetons dans la place $Activ_0$ (voir figure 8.6).
   Nous complétons donc la **fonction de valuation** par :
   (a) $W(Time(0), Clk_0) = P$;
   (b) $W(Clk_0, Activ_0) = P(1 - U_{max})$;

2. sinon, si $P(1 - U_{max}) < 0$,
   il faut tenir compte du retard avant le dépôt de jetons dans la place $Oisiv_0$ (voir figure 8.7).
   Nous complétons dans ce cas la **fonction de valuation** par :
   (a) $W(Time(0), Clk_0) = W(Time(0), Clk'_0) = P$;
   (b) $W(Clk_0, Retard) = d = |P(1 - U_{max})|$;
   (c) $W(Debut, Activ_0) = v$, où :
      i. dans l'approche chemin, *Debut* correspond à la première transition du comportement choisi de durée $C_{imax} - v$;
      ii. dans l'approche arbre, *Debut* correspond à la première transition du sous comportement choisi, telle que la différence entre la durée d'exécution si on avait choisi la plus longue branche et la durée d'exécution de la plus longue branche de l'embranchement choisi soit v.

   Cet aspect est illustré sur les figures 8.14 et 8.15.

Notons pour conclure, que la modélisation de la couche temporelle est identique, que l'on utilise l'approche chemin ou l'approche arbre.

8.1.3 Modélisation de la couche sémantique

Nous avons supposé que les tâches considérées étaient sémantiquement correctes (voir définition 4.3.1).
L'aspect sémantique concerne donc ici uniquement la prise en compte des relations d'incompatibilité entre les différents comportements des tâches de l'application.
Nous distinguons deux approches, suivant qu'on considère qu'une tâche est représentée comme un ensemble de chemins, ou comme un arbre.

8.1.3.1 Approche chemin

Notre objectif est de garantir l'exclusion mutuelle entre deux comportements incompatibles, tout en assurant que tous les comportements cohérents restent préservés.
Nous voulons donc implémenter $\mathcal{RIT}$ dans le modèle.

Pour chaque couple $(Comp_{il}, Comp_{i'l'})_{i<i'} \in \mathcal{RI}$ de comportements incompatibles, nous créons une place $Excl_{ili'l'}$ (pour la gestion de l'exclusion mutuelle) qui contient initialement un jeton, et qui est reliée aux premières transitions de ces comportements.
Le tir d'une de ces transitions détermine le comportement choisi, et un seul comportement peut être choisi à la fois, grâce au jeton de la place $Excl_{ili'l'}$ qui met les transitions T_{i11l} et $T_{i'11l'}$ en exclusion mutuelle.
Les deux tâches T_i et $T_{i'}$ sont réactivées simultanément (voir définition 5.2.1).
À chacune de leur réactivation, il faut s'assurer que $Excl_{ili'l'}$ contient bien un et un

seul jeton.
Si lors des instances précédentes, les comportements choisis n'étaient ni $Comp_{il}$ ni $Comp_{i'l'}$, la place $Excl_{ili'l'}$ est déjà marquée, sinon il faut y déposer 1 jeton lors de la réactivation.
Pour cela, nous associons à chaque comportement $Comp_{il}$ une place Sem_{il} (pour la gestion de la cohérence sémantique) initialement vide.
Cette place permettra d'identifier le comportement qui a été exécuté, et de réinitialiser les places d'exclusion mutuelle qui lui sont associées.
Le tir de T_{i11l} y dépose un jeton.
Nous donnons figure 8.11 une illustration de la modélisation de l'exclusion mutuelle.

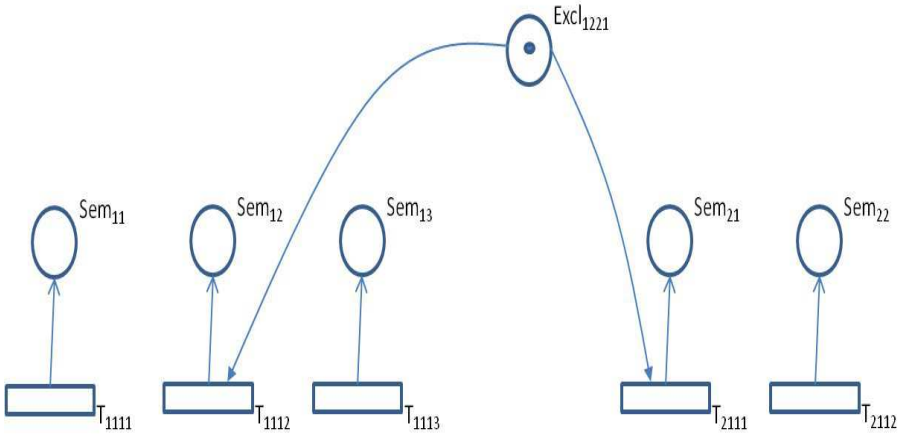


Figure 8.11 – Modélisation de l'exclusion mutuelle : les comportements $Comp_{12}$ et $Comp_{21}$ sont incompatibles

Nous remplaçons ensuite la transition initiale Clk_i qui était associée à la tâche T_i, par un ensemble de transitions $\{Clk_{il},\ l \in 1, \ldots, Compeff_i\}$, chacune associée au comportement $Comp_{il}$.
Le tir de Clk_{il} nécessite P_i jetons dans $Time(i)$ et un (1) jeton dans Sem_{il}, et il dépose un (1) jeton dans toutes les places d'exclusion en entrée de T_{i11l}. Il s'agit des places $Excl_{il..}$ et $Excl_{..il}$.

Nous illustrons figure 8.12 la gestion complète de la cohérence sémantique.

Dans le cas où les dates de réveil sont nulles, l'activation des tâches se fera sans souci, puisque les places $Activ_i$ ont initialement $\mathbf{a}+\mathbf{b}$ jetons.
Si les dates de réveil de certaines tâches sont non nulles, les places $Activ_i$ de ces tâches contiennent initialement $\mathbf{b}$ jetons, et les places Sem_{il} 0 jeton.
Pour qu'une première activation soit possible, nous complétons la couche sémantique en ajoutant à chaque tâche T_i de date de premier réveil non nulle une place

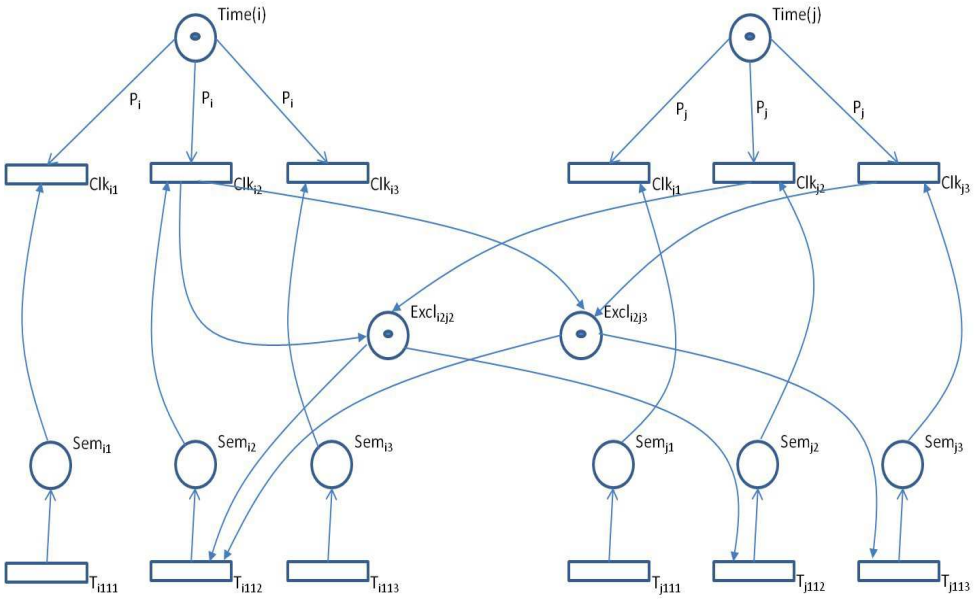


Figure 8.12 – Gestion de la cohérence sémantique : les couples de comportements $(Comp_{i2}, Comp_{j2})$ et $(Comp_{i2}, Comp_{j3})$ sont incompatibles

Sem_{i0}, qui contient initialement 1 jeton, et une transition Clk_{i0}.

Nous donnons figure 8.13 une illustration de ce principe.

Nous complétons **la fonction de valuation** :

1. $W(Excl_{ili'l'}, T_{i11l}) = W(Excl_{ili'l'}, T_{i'11l'}) = 1$;
2. $W(Clk_{il}, Excl_{ili'l'}) = W(Clk_{i'l'}, Excl_{ili'l'}) = 1$;
3. $W(T_{i11l}, Sem_{il}) = W(Sem_{il}, Clk_{il}) = 1$;
4. $W(Clk_{il}, Activ_i) = \mathbf{a}$;
5. $W(Time(i), Clk_{i0}) = W(Time(i), Clk_{il}) = P_i$, $l \in 1, \ldots, Compeff_i$.

De même, nous ajoutons au **marquage initial** :

1. $M_0(Sem_{il}) = 0$ pour $l > 0$;
2. $M_0(Sem_{i0}) = 1$;
3. $M_0(Excl_{ili'l'}) = 1$.

Pour conclure cette section sur la modélisation des applications temps-réel en utilisant notre modèle de réseau de Petri, nous donnons figure 8.14 la modélisation complète du système de contrôle des essuie-glaces de l'exemple 1.2.2.1 suivant l'approche chemin.
Remarquons que nous sommes dans le cas $P(1-U_{max}) < 0$. La tâche oisive est donc

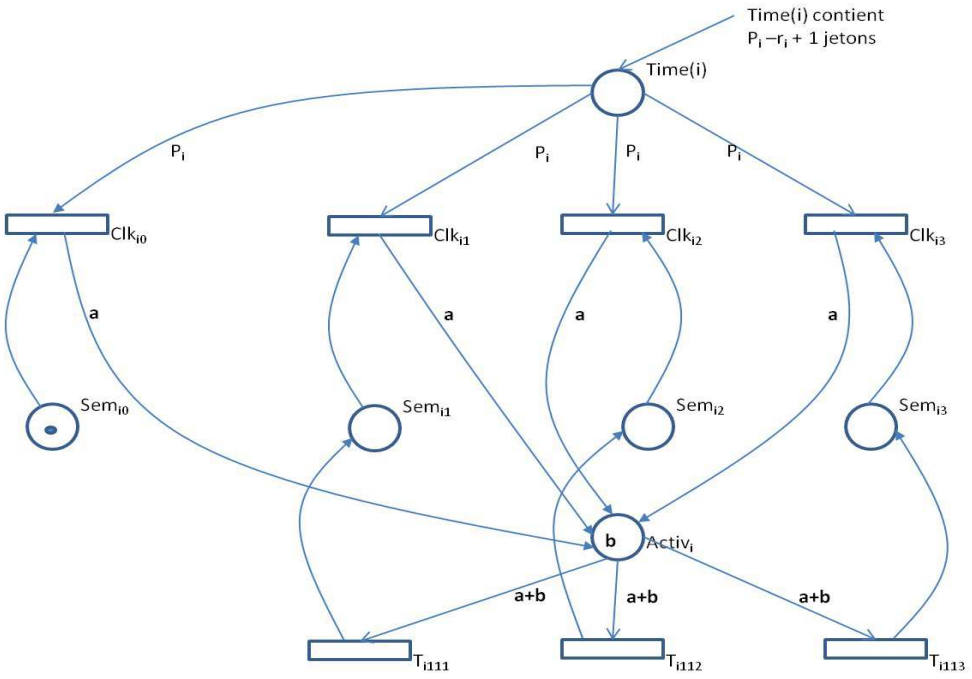


Figure 8.13 – Complétion de la couche sémantique pour une tâche T_i quand $r_i > 0$

représentée suivant la figure 8.7.
Nous pouvons aussi remarquer le dépôt de 2 jetons par la transition T_{2111} dans la place $Activ_0$ lorsque la tâche T_2 choisi le chemin de durée 4, qui correspond à $6-2$.
Par souci de lisibilité, les 5 arcs valués par **b** qui relient les dernières transitions aux places $Activ_i$ $(i \geq 1)$ n'ont pas été représentés.
De même, la place *Processor* et les 30 arcs qui la relient à chaque transition de la couche structurelle ne sont pas représentés.
Les tâches étant à départs simultanés, les places Sem_{i0} n'existent pas dans ce cas.
Notons enfin que la place Sem_{11}, comme toutes les places *Sem* des tâches qui ne possèdent pas d'instruction conditionnelle, ou bien dont les comportements ne sont incompatibles avec aucun autre comportement d'autres tâches, est représentée par souci d'uniformisation.
Nous aurions pu en effet pour la tâche T_1, comme pour les autres types de tâches que nous venons de citer, garder la représentation de [Pailler 2006].

8.1.3.2 Approche arbre

L'idée générale consiste à garder la même philosophie que dans l'approche chemin pour la gestion de la cohérence sémantique, à la différence que dans la gestion des exclusions mutuelles, les places d'exclusion ne portent pas sur tout le comportement, mais uniquement sur les *sous comportements* correspondants.
Nous implémentons dans cette approche $\mathcal{RIT}_{min}$.

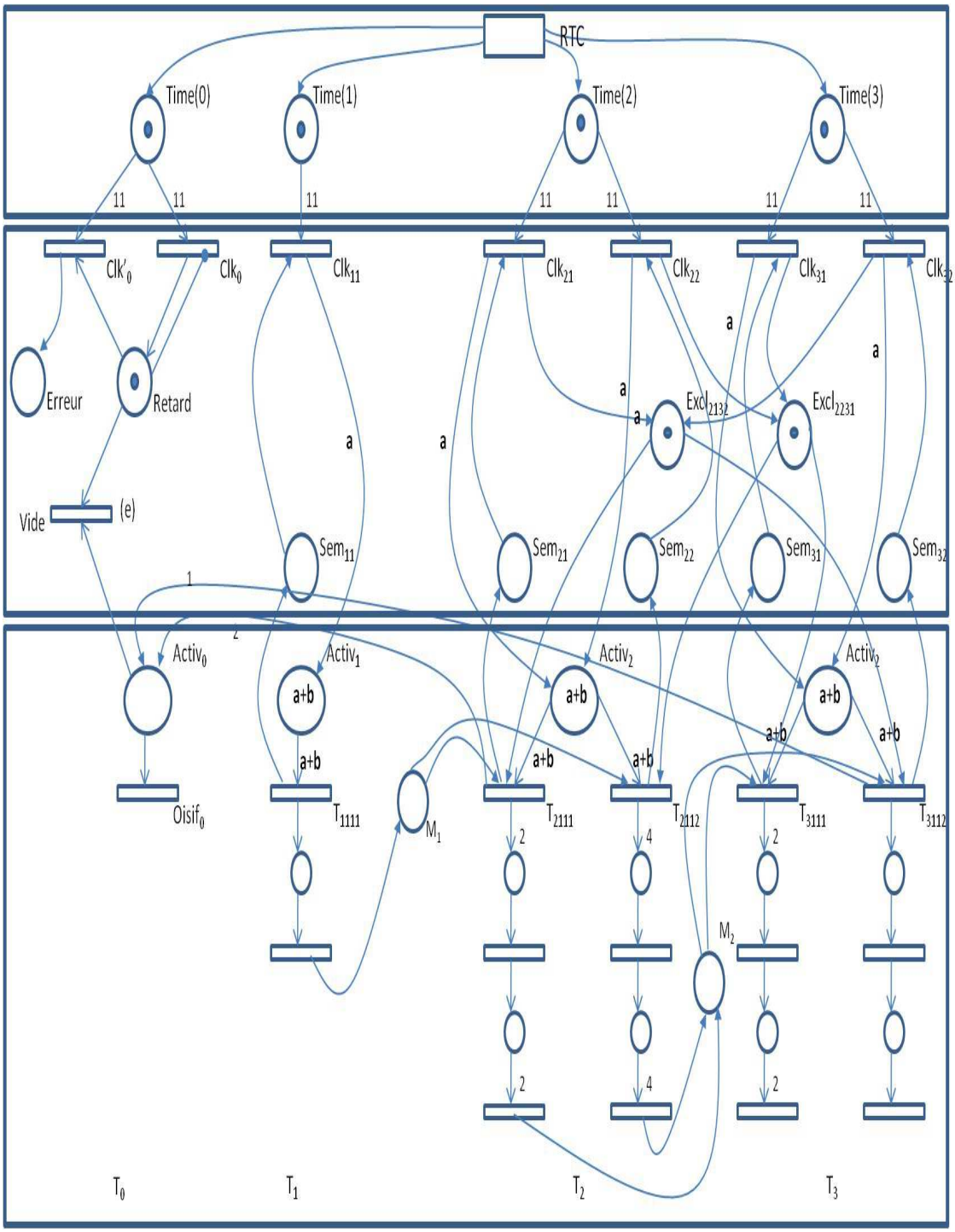


Figure 8.14 – Modélisation, par réseau de Petri, du système des essuie-glaces suivant l'approche chemin (ici, $P(1 - U_{max}) < 0$)

Nous allons donner une modélisation formelle de la couche sémantique dans l'approche arbre.

Afin de lier un comportement d'une tâche à l'ensemble des tests qui l'induit, nous notons $Comp_{iL}$, L étant définit section 8.1.1.2, le comportement de la tâche T_i qui est induit par la liste des tests et signes contenus dans L.
Pour obtenir l'ensemble des comportements d'une tâche dans l'approche arbre, il faut donc se référer à L pour la dernière transition sur chaque sous comportement.

Par exemple, pour la tâche T_1 de la figure 8.5, l'ensemble de ses comportements est $\{Comp_{1((t_1,+,2))},\ Comp_{1((t_1,-,5)(t_2,+,3))},\ Comp_{1((t_1,-,4)(t_2,-,2))}\}$.
De même, pour la tâche T_2 de la figure 8.5, l'ensemble de ses comportements est $\{Comp_{2((t_3,+,3))},\ Comp_{2((t_3,-,3))}\}$.

Pour chaque couple $(Comp_{iL}, Comp_{i'L'})_{i<i'} \in \mathcal{RI}_{min}$, nous savons qu'il existe au moins $(t,s,p) \in L$, $(t',s',p') \in L'$ tel que $(t,s)\ \mathcal{RIT}_{min}\ (t',s')$ (cette notation est un abus, et signifie que les tests t et t', respectivement les résultats suivants s et s', sont incompatibles, et appartiennent à la relation d'incompatibilité minimale, c'est à dire $t^s\ \mathcal{RIT}_{min}\ t'^{s'}$).
Pour chacun de ces couples de tests, nous créons une place d'exclusion mutuelle $Excl_{i(t,s,1)i'(t',s',1)}$, qui contient initialement 1 jeton, qui met en exclusion mutuelle les sous comportements induits par les tests t (suivant s) et t' (suivant s').
Ces places d'exclusion mutuelle sont reliées aux premières transitions des sous comportements correspondants. Ce qui explique le 1 à la place de p, et le 1 à la place de p'.

Ensuite, comme dans l'approche chemin, nous gérons la cohérence sémantique.
Pour chaque comportement $Comp_{iL}$, nous créons une place Sem_{iL} qui contient initialement 0 jeton.
Un jeton y est déposé quand on tire la dernière transition de ce comportement, c'est à dire en même temps qu'on dépose le jeton **b** dans la place $Activ_i$.
Le tir de l'horloge locale correspondante (pour une tâche il y a autant d'horloges locales que de comportements) remettra alors des jetons dans toutes les places d'exclusions mutuelles associées à ce comportement.

Enfin, comme dans l'approche chemin, nous modifions l'horloge locale Clk_i, en la remplaçant pour chaque tâche T_i par un ensemble d'horloges Clk_{iL}, L appartenant aux dernières transitions de chaque comportement.
Il y a donc exactement autant d'horloges locales Clk_{iL} que de comportements $Comp_{iL}$.

Nous complétons dans l'approche arbre la **fonction de valuation** :

1. $W(Excl_{i(t,s,1)i'(t',s',1)}, T_{ij1(\ldots(t,s,1)\ldots)}) = 1$;

2. $W(Excl_{i(t,s,1)i'(t',s',1)}, T_{i'j'1(...(t',s',1)...)}) = 1$;
3. $W(T_{ijkL}, Sem_{iL} = 1$, où L appartient aux dernières transitions de chaque comportement ;
4. $W(Sem_{iL}), Clk_{iL}) = 1$;
5. $W(Clk_{iL}, Excl_{i(t,s,1)i'(t',s',1)}) = 1$ si $(t, s, 1) \in L$;
6. $W(Clk_{i'L'}, Excl_{i(t,s,1)i'(t',s',1)}) = 1$ si $(t', s'', 1) \in L'$;
7. $W(Time(i), Clk_{iL}) = P_i$;
8. $W(Clk_{iL}, Activ_i) = \mathbf{a}$.

De même nous ajoutons au **marquage initial** :

1. $M_0(Excl_{i(t,s,1)i'(t',s',1)}) = 1$;
2. $M_0(Sem_{iL})) = 0$.

Nous donnons figure 8.15 la modélisation complète du système de contrôle des essuie-glaces de l'exemple 1.2.2.1 suivant l'approche arbre.
Lorsque la tâche T_2 choisit le sous comportement qui commence par la transition $T_{231((t_1,+,1))}$, la différence entre la plus longue branche au niveau du test et la plus longue branche du comportement choisi (ici c'est le comportement lui même) vaut $3 - 1 = 2$. C'est la raison pour laquelle 2 jetons sont déposés dans $Activ_0$.
On utilise le même raisonnement pour déposer 1 jeton dans $Activ_0$ lorsqu'on tire la transition $T_{321((t_2,-,1))}$
La place $Sem_{1()}$ existe par souci d'uniformisation, ainsi que la notation $Clk_{1()}$ utilisée pour l'horloge locale de la tâche T_1.

8.2 Validation du modèle

Nous présentons ici la validation par l'approche chemin.
L'approche arbre étant une factorisation de l'approche chemin, la validation sur l'approche arbre pourra se déduire de celle sur l'approche chemin.
Nous souhaitons montrer que :

1. nous ne pouvons pas exécuter dans une même période deux chemins incompatibles ;
2. aucun comportement n'est définitivement perdu : à chaque activation, chaque tâche dispose bien de tous ses chemins effectifs.

Pour cela, nous énonçons deux invariants qui doivent être vérifiés par notre système.

Invariant 8.2.1 $\forall\ i, i' \in 1 \ldots n$, $\forall\ l \in 1 \ldots |Compeff_i|$, $\forall\ l' \in 1 \ldots |Compeff_{i'}|$, $M(Sem_{il}) + M(Excl_{ili'l'}) + M(Sem_{i'l'}) = 1$.

L'invariant 8.2.1 assure que deux chemins incompatibles ne peuvent pas être choisis pendant la même période.

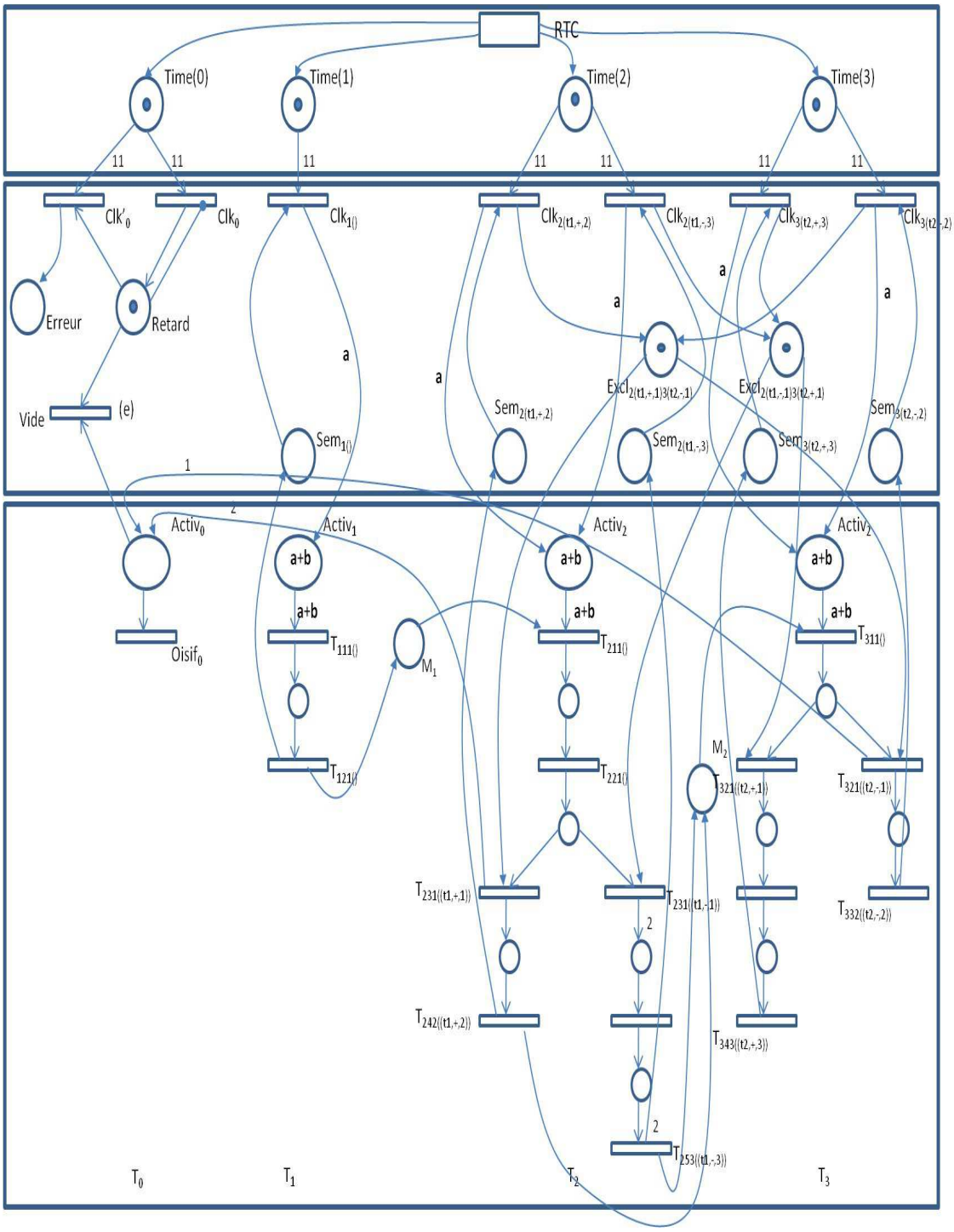


Figure 8.15 – Modélisation, par réseau de Petri, du système des essuie-glaces suivant l'approche arbre (ici, $P(1 - U_{max}) < 0$). Nous supposons que $t_1 = (contact = 0)_{T_2}$ et $t_2 = (contact = 0)_{T_3}$

Preuve 8.2.1 Nous faisons une preuve par induction sur les marquages accessibles.
L'invariant est vérifié pour M_0 car $M_0(Sem_{il}) = M_0(Sem_{i'l'}) = 0$ et $M_0(Excl_{ili'l'}) = 1$.
Supposons que l'invariant soit vrai pour un marquage accessible M quelconque.
Soit σ un ensemble maximal de transitions tiré depuis M, qui fait passer le système dans un état M'.
Notre objectif est de montrer que l'invariant reste vrai dans l'état M'.
Soit $Trans = \{T_{i11l}, T_{i'11l'}, Clk_{il}, Clk_{i'l'}\}$ l'ensemble des transitions en entrée ou en sortie des places qui interviennent dans l'invariant (donc qui sont susceptibles de modifier l'invariant).
Une seule de ces transitions peut être tirée à la fois :

1. si $\sigma \cap Trans = \emptyset$, le tir de σ ne modifie pas le marquage de ces places. L'invariant reste donc valide ;
2. sinon, plusieurs cas sont possibles :
   (a) si $T_{i11l} \in \sigma$, par précondition, nous avions $M(Excl_{ili'l'}) > 0$ (donc $= 1$). Par suite, $M(Sem_{il}) = M(Sem_{i'l'}) = 0$, donc ni Clk_{il} ni $Clk_{i'l'}$ ne sont tirables.
   De même $T_{i'11l'}$ n'est pas franchissable en même temps que T_{i11l}, car la place *Processor* met toutes les transitions du système de tâches en exclusion mutuelle.
   À l'issue du tir, la marque de $Excl_{ili'l'}$ est consommée, donc $M'(Excl_{ili'l'}) = 0$.
   Une marque est déposée dans Sem_{il} donc $M(Sem_{il}) = 1$ et le marquage de $Sem_{i'l'}$ n'est pas modifié.
   Donc l'invariant reste valide ;
   (b) si $T_{i'11l'} \in \sigma$, nous utilisons le même raisonnement qu'au point 2a ;
   (c) si $Clk_{il} \in \sigma$, on avait 1 marque dans Sem_{il}, 0 dans $Excl_{ili'l'}$ donc ni T_{i11l} ni $T_{i'11l'}$ ne sont franchissables, et 0 marque dans $Sem_{i'l'}$, donc $Clk_{i'l'}$ n'est pas franchissable.
   Le tir de Clk_{il} consomme la marque de Sem_{il}, et en dépose 1 dans $Excl_{ili'l'}$.
   Comme le marquage de $Sem_{i'l'}$ n'est pas modifié, l'invariant reste valide ;
   (d) si $Clk_{i'l'} \in \sigma$, nous utilisons le même raisonnement qu'au point 2c.

Invariant 8.2.2 $\forall\, i \in 1 \ldots n$, $\forall\, l \in 1 \ldots |Compeff_i|$, $\sum_{l=1}^{|Compeff_i|} M(Sem_{il}) \leq 1$ et $\forall\, i \in 1 \ldots n$, $\mathrm{M}(Activ_i) = \mathbf{a}{+}\mathbf{b} \Rightarrow \forall\, l \in 1 \ldots |Path_i|$, $\mathrm{M}(Sem_{il}) = 0$.

L'invariant 8.2.2 garantit qu'à l'issue de la réactivation, aucun chemin n'est marqué comme étant en exécution, et, combiné avec l'invariant 8.2.1, toutes les places d'exclusion mutuelle sont bien marquées.
Donc, tous les chemins sont bien possibles.

Preuve 8.2.2 Comme précédemment, nous faisons une preuve par induction.
Par définition $\sum_{l=1}^{|Compeff_i|} M_0(Sem_{il}) = 0$.

Nous supposons maintenant que l'invariant est vérifié pour un marquage M quelconque.
Soit σ un ensemble maximal de transitions tiré, qui fait passer le système dans un état M'.
Montrons que les propriétés restent valides dans l'état M'.
Soit $Trans = \{T_{i11l}, Clk_{il}\}$ l'ensemble des transitions en entrée ou en sortie des places qui interviennent dans l'invariant (donc qui sont susceptibles de modifier l'invariant) :

1. si $\sigma \cap Trans = \emptyset$, le tir de σ ne modifie le marquage d'aucune place Sem_{il}.
   Les propriétés restent donc valides ;
2. sinon, deux cas sont possibles :
   (a) M($Activ_i$) = **a**+**b**.
   Dans ce cas, par hypothèse, toutes les places Sem_{il} sont vides, et donc seule T_{i11l} peut être tirée.
   À l'issue du tir, la place Sem_{il} est marquée, les autres restent vides.
   Donc $\sum_{l=1}^{|Compeff_i|} M'(Sem_{il}) = 1$;
   (b) M($Activ_i$) $\neq$ **a**+**b**.
   Dans ce cas, à cause de l'ensemble terminal, $Activ_i$ ne contient pas **a**, donc seule Clk_{il} peut être tirée.
   À l'issue du tir, le jeton (unique par hypothèse de récurrence) de Sem_{il} est consommé.
   Par conséquent, $\sum_{l=1}^{|Compeff_i|} M'(Sem_{il}) = 0$. □

Nous ajoutons à ces invariants la propriété 8.2.1, qui garantit qu'à l'issue de la réactivation, toutes les places d'exclusion mutuelle sont marquées.

Propriété 8.2.1 $\forall\, i \in 1 \ldots n, \forall\, l \in 1 \ldots |Compeff_i|, \forall l' \in 1 \ldots |Compeff_{i'}|$, $M(Time(i)) = 1 \Rightarrow M(Excl_{ili'l'}) = 1$.

Preuve 8.2.3 Soit $Excl_{ili'l'}$ une place d'exclusion mutuelle.
Si $M(Time(i)) = 1$, par définition de la relation d'incompatibilité, nous avons $r_i = r_{i'}$ et $P_i = P_{i'}$, donc $M(Time(i')) = 1$.
De même, si on vient de réactiver la tâche T_i (en tirant une transition Clk_{ik}), on a réactivé la tâche $T_{i'}$ (en tirant une transition $Clk_{i'k'}$).
Soit M' le marquage qui précède :

1. si $k = l$, on avait $M'(Sem_{il}) = 1$, donc $M'(Sem_{i'l'}) = M'(Excl_{ili'l'}) = 0$ (invariant 8.2.1).
   Ainsi, à l'issue du tir de Clk_{il}, on a consommé la marque de Sem_{il} et déposé une (1) marque dans $Excl_{ili'l'}$, donc $M(Excl_{ili'l'}) = 1$;
2. si $k \neq l$, Clk_{ik} ne dépose pas de marques dans $Excl_{ili'l'}$, et on a :
   $M'(Sem_{ik}) = 1$, donc $M'(Sem_{il}) = 0$ (invariant 8.2.2) :
   (a) si $k' = l'$, $Clk_{i'l'}$ transfère une (1) marque de $Sem_{i'l'}$ dans $Excl_{ili'l'}$, donc $M(Excl_{ili'l'}) = 1$;

(b) si $k' \neq l'$, $Clk_{i'l'}$ ne dépose pas de marque dans $Excl_{ili'l'}$, et on avait $M'(Sem_{i'l'}) = 0$.
Dans ce cas, on a $M'(Excl_{ili'l'}) = 1$ (invariant 8.2.1), et les tirs de Clk_{ik} et $Clk_{i'k'}$ n'ont pas modifié le marquage de $Excl_{ili'l'}$. □

Remarquons que nos objectifs sont atteints :

1. deux comportements incompatibles $Comp_{il}$ et $Comp_{i'l'}$ $(i < i')$ ne peuvent pas avoir été exécutés pendant la même période, sinon il existerait un marquage M tel que $M(Sem_{il}) = M(Sem_{il'}) = 1$.
   Ce qui contredit l'invariant 8.2.1 ;
2. à l'issue de la réactivation, tous les comportements d'une tâche peuvent être choisis, car $M(Excl_{ili'l'}) = 1, \forall\ i, i', l, l'$ (propriété 8.2.1).

En conclusion, l'exclusion mutuelle entre chemins incompatibles est bien respectée, et aucun comportement n'est perdu.
Nous avons donc validé le modèle de réseau de Petri que nous avons proposé.
Nous nous intéressons section 8.3 à l'obtention d'un tel modèle.
Nous donnerons chapitre 9 une vue générale de l'utilisation de ce modèle.

8.3 Obtention du modèle

L'objectif de cette section est de montrer, d'un point de vue implémentation, comment en partant du code source d'un système de tâches, nous obtenons notre modèle de réseau de Petri.
Nous présentons donc ici notre méthodologie d'obtention des modèles de réseaux de Petri approche arbre et approche chemin.
Les tâches que nous manipulons sont écrites en pseudo code C.
Nous donnons annexe .3 la grammaire **BNF** de ces tâches, et figure 8.16 une vue graphique simplifiée de la grammaire de l'annexe .3.
Les systèmes manipulés ont donc un nom, éventuellement une liste de ressources, de messages et de variables, et possèdent au moins deux tâches.
Chaque tâche est caractérisée par un nom, une date de premier réveil, un délai critique, une période, éventuellement une liste de variables, et une liste d'instructions.
Les instructions peuvent être des instructions de prise de ressources, de libération de ressources, d'envoi de messages, de réception de messages, ou bien des instructions conditionnelles.
Les instructions conditionnelles sont constituées de tests (qui ont une durée), d'une liste d'instructions correspondant au *Then*, et éventuellement d'une liste d'instructions correspondant au *Else*.
Chaque test est composé d'une variable, d'un opérateur de test (qui peut être $<$, $\leq$, $>$, $\geq$, $=$ ou $\neq$) et de la valeur de la variable qui dépend du type de la variable.
Nous avons en effet dans cette étude restreint l'ensemble des tests aux seuls tests de comparaison.

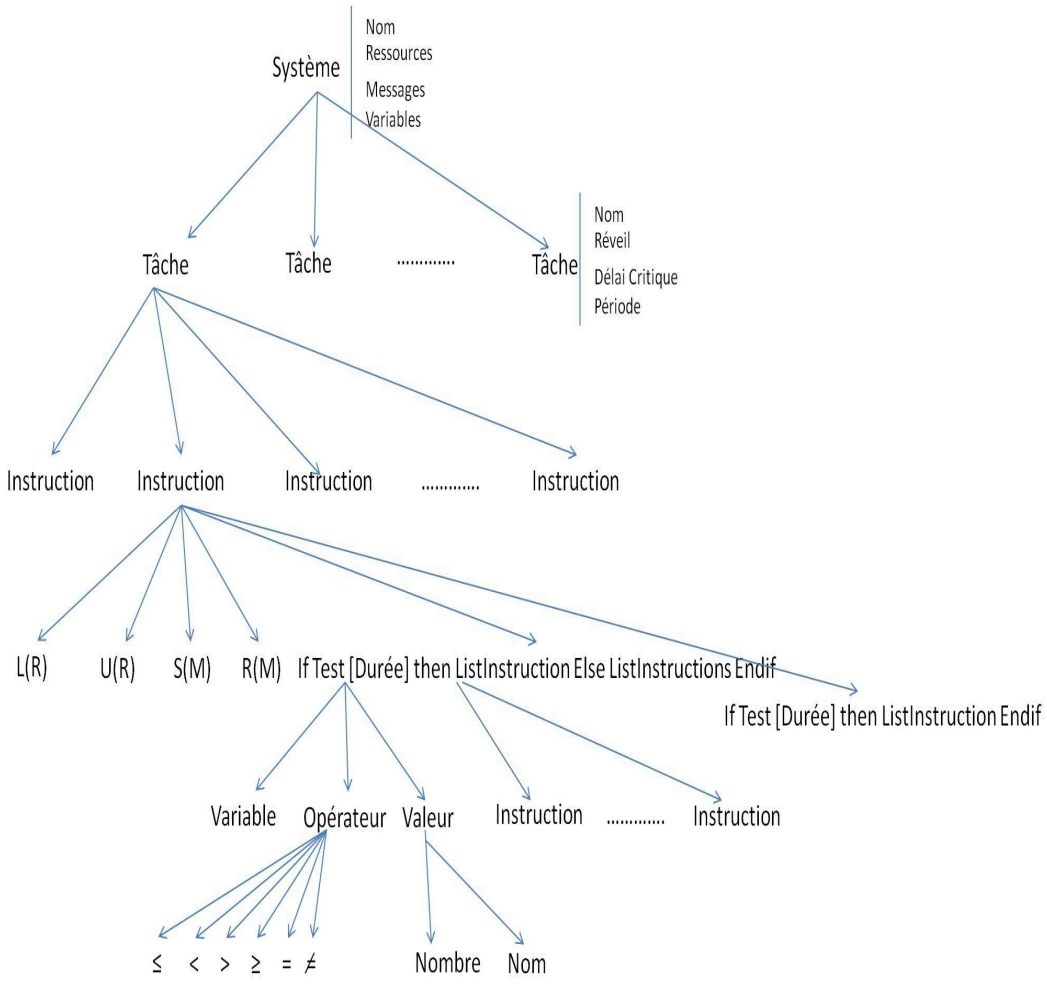


Figure 8.16 – Vue graphique simplifiée de la grammaire **BNF** des tâches manipulées

Cette grammaire est utilisée dans l'outil **PETRIGEN** (en cours de développement) par les analyseurs syntaxiques et sémantiques **FLEX** [Vern 1995] et **BISON** [Donnelly 1995] pour l'obtention de l'ensemble des relations d'incompatibilité et des modèles de réseau de Petri approche chemin et arbre.
Une vue générale de **PETRIGEN** est donnée figure 8.17.

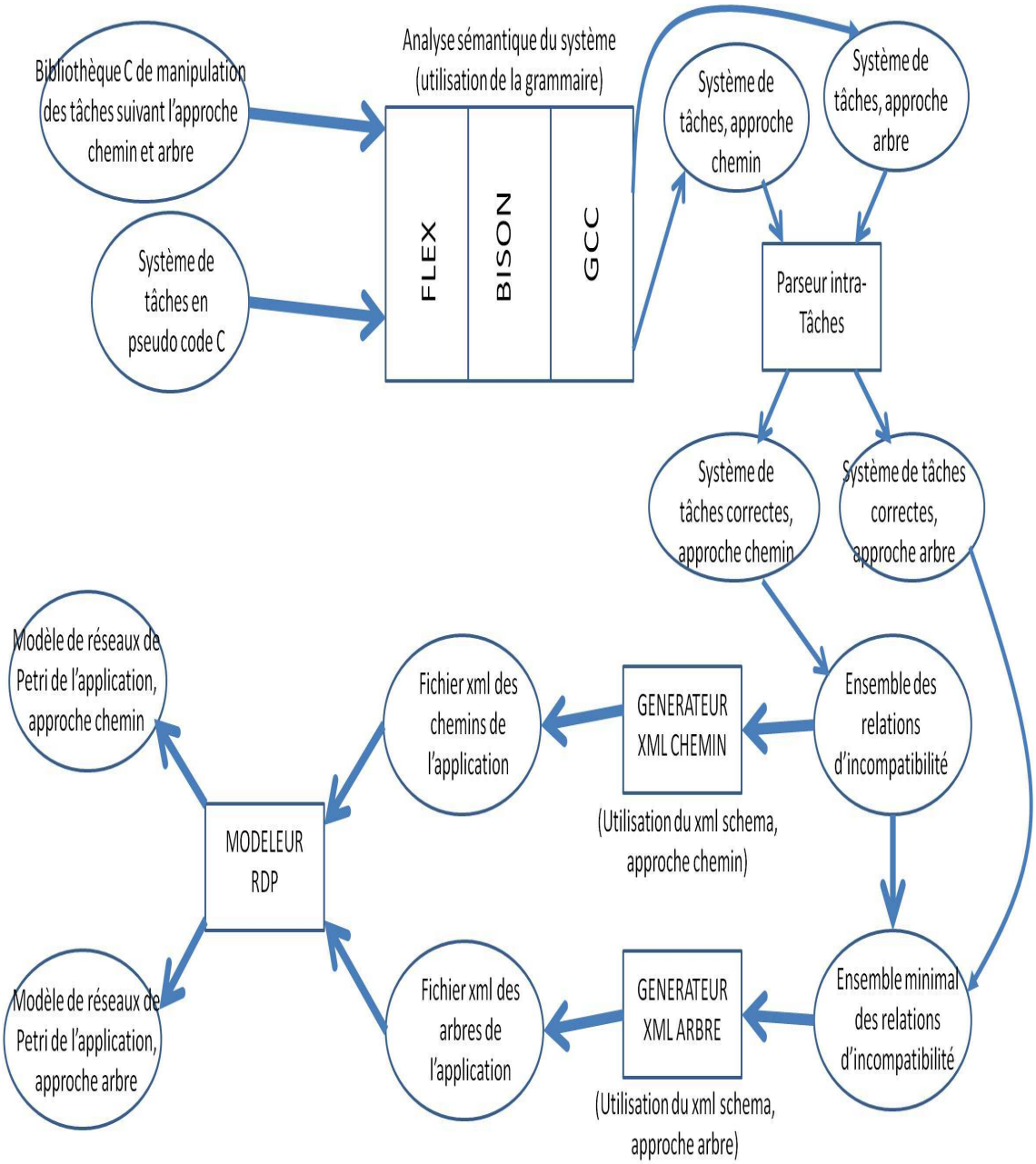


Figure 8.17 – Architecture de **PETRIGEN**

L'outil **PETRIGEN** prend en entrée le système de tâches, et fournit en sortie les modèles de réseau de Petri approche chemin et arbre (sémantiquement corrects) que nous avons présenté.
Une première étape consiste à représenter les tâches en pseudo C en entrée, suivant l'approche chemin ou l'approche arbre.
Ensuite, un **Parseur Intra-Tâches** permettra d'obtenir le système de tâches correctes suivant l'approche chemin ou l'approche arbre, en éliminant les chemins qui possèdent des tests incompatibles (voir définition 4.2.3).
C'est à partir de ces systèmes corrects que nous allons construire les modèles de réseaux de Petri :

1. si l'objectif est de construire le modèle de réseau de Petri approche chemin, nous utilisons $\mathcal{RIT}$.
   Le principe de construction de $\mathcal{RIT}$ est donné méthode 8.3.1.

   Méthode 8.3.1 Construction de $\mathcal{RIT}$

   (a) Pour $i \in 1, \ldots, n$, où n est le nombre de tâches de l'application

   i. Pour $j \in i+1, \ldots, n$

   A. Pour $t_1 \in \mathcal{R}esTestPar_i$, pour $t_2 \in \mathcal{R}esTestPar_j$ si $t_1 \ \mathcal{RIT} \ t_2$ alors $\mathcal{RIT} := \mathcal{RIT} \cup \{(t_1, t_2)\}$.

   Une fois l'ensemble des relations d'incompatibilité construit, le **GENERATEUR XML CHEMIN** va prendre en entrée l'ensemble des tâches de l'application, modélisées suivant l'approche chemin, et construire le fichier XML des chemins des tâches et des chemins incompatibles de l'application, dont le xml schema [Sperberg-McQueen] est donné en annexe .4.
   Nous donnons figure 8.18 une vue graphique simplifiée du xml schéma de l'annexe .4.
   Un système est cette fois vu comme un ensemble de tâches, chacune représentée comme un ensemble de chemins, et un ensemble de configurations de chemins incompatibles, déterminés en vérifiant s'ils possèdent au moins deux tests qui sont incompatibles.
   Les chemins sont constitués d'instructions qui peuvent être des blocs généraux, des primitives temps-réel ou bien des tests.
   C'est à partir de ces tests (contenus dans $\mathcal{RIT}$) qu'on déterminera les chemins incompatibles.
   Chaque configuration impossible (il y a en fait autant de configurations impossibles que d'éléments de $\mathcal{RIT}$) est constituée de 2 couples.
   Chaque couple contient la tâche (en fait le numéro de la tâche) et le chemin (en fait le numéro du chemin) correspondant qui intervient dans la relation d'incompatibilité.

   Le **MODELEUR RDP** prend en entrée des fichiers XML qui respectent le XML schéma de l'annexe .4 et construit le modèle final du réseau de Petri

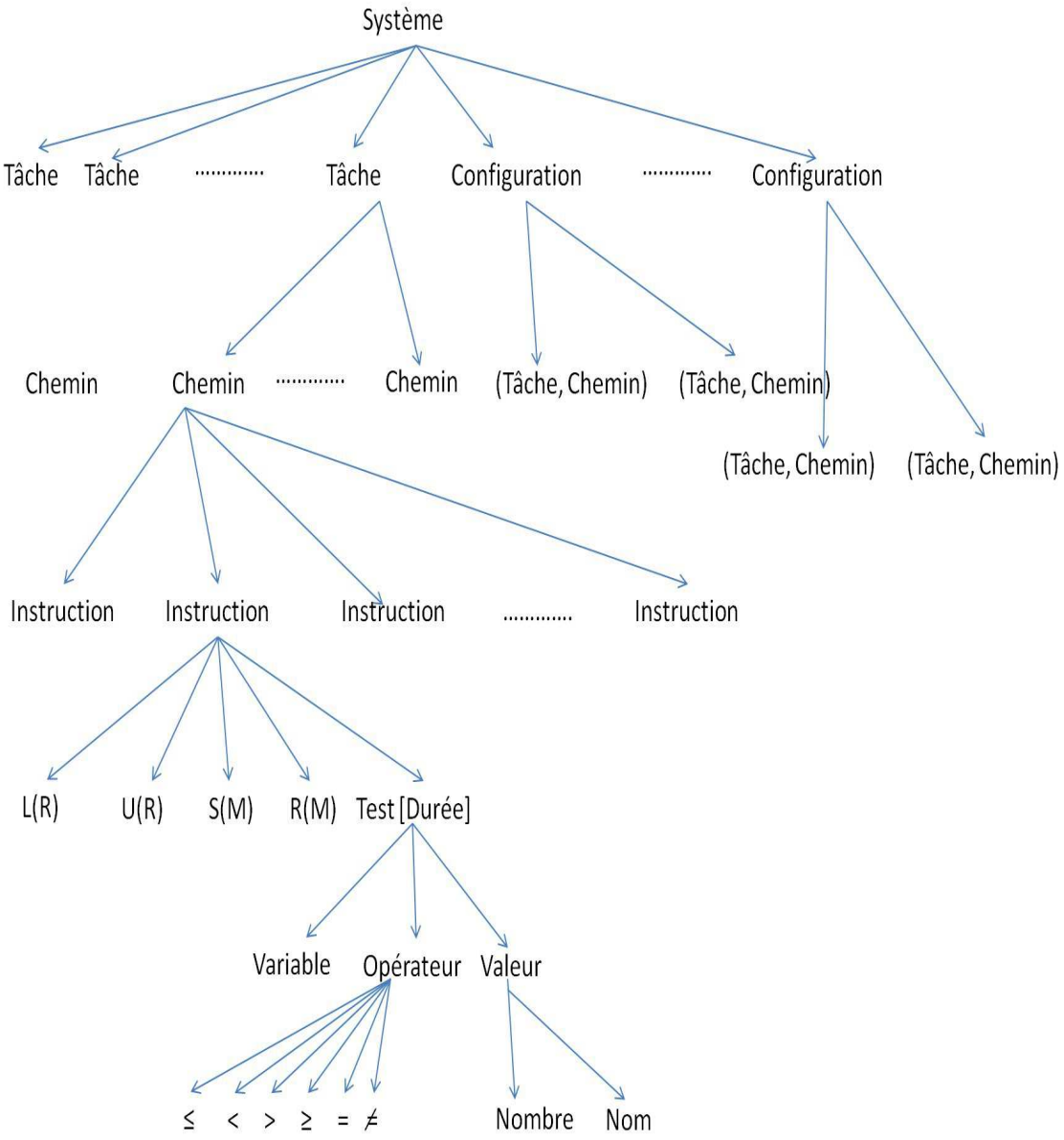


Figure 8.18 – Vue graphique simplifiée du XML Schéma du fichier permettant la construction du modèle de réseau de Petri, approche chemin, des applications

suivant l'approche chemin.
Il faudra penser à ajouter éventuellement la tâche oisive ;

2. si l'objectif est de construire le modèle de réseau de Petri approche arbre, nous utilisons $\mathcal{RIT}_{min}$.
Il se déduit de $\mathcal{RIT}$ en utilisant méthode 5.2.1.

Une fois l'ensemble des relations d'incompatibilité minimal obtenu, le **GENERATEUR XML ARBRE** va prendre en entrée l'ensemble des tâches de l'application, modélisées suivant l'approche arbre, et construire le fichier XML des arbres des tâches en tenant compte des branches incompatibles, dont le xml schema est donné en annexe .5.
Nous donnons figure 8.19 une vue graphique simplifiée du xml schéma de l'annexe .5.
Un système est, comme dans l'approche chemin, toujours constitué d'un ensemble de tâches et de configurations impossibles.
Une tâche est dans ce cas vue comme un arbre.
Les arbres sont constitués d'un ensemble de nœuds, et la structure des nœuds est celle décrite dans l'annexe .2.
Un nœud est donc composé d'une racine (qui est identifiée de façon unique, et qui contient la durée d'exécution de la tâche depuis son activation), de deux étiquettes (l'étiquette 1 est celle utilisée couramment, et l'étiquette 2 sert à prendre en compte la partie *Else* du test en présence d'une instruction conditionnelle) et de deux fils (le fils 1 est utilisé couramment et pointe sur le nœud suivant, et le fils 2 est utilisé chaque fois qu'on a l'étiquette 2 et pointe sur le nœud correspondant).
Enfin, chaque étiquette est composée du numéro de la tâche (ici nous avons gardé l'appellation *Mère*), de la nature de l'étiquette (*Nat*), du numéro de l'étiquette (*Num*) et de l'entier complément (*Compl*) dont la sémantique dépend de la nature de l'étiquette.
L'étiquette peut être a (pour instruction élémentaire de durée unitaire), t (pour test), L (pour la prise d'une ressource), U (pour la libération d'une ressource), S (pour l'envoi d'un message), R (pour la réception d'un message), V (pour signifier qu'il ne se passe rien).
Chaque configuration impossible (il y a autant de configurations impossibles que d'éléments de $\mathcal{RIT}_{min}$) est composée de 2 couples.
Chaque couple contient le nœud de la tâche (caractérisé par sa *racine*) et l'étiquette correspondante qui intervient dans la relation d'incompatibilité.

Le **MODELEUR RDP** prend en entrée des fichiers XML qui respectent le XML schéma de l'annexe .5 et construit le modèle final du réseau de Petri suivant l'approche arbre.
Il faudra penser à ajouter éventuellement la tâche oisive ;

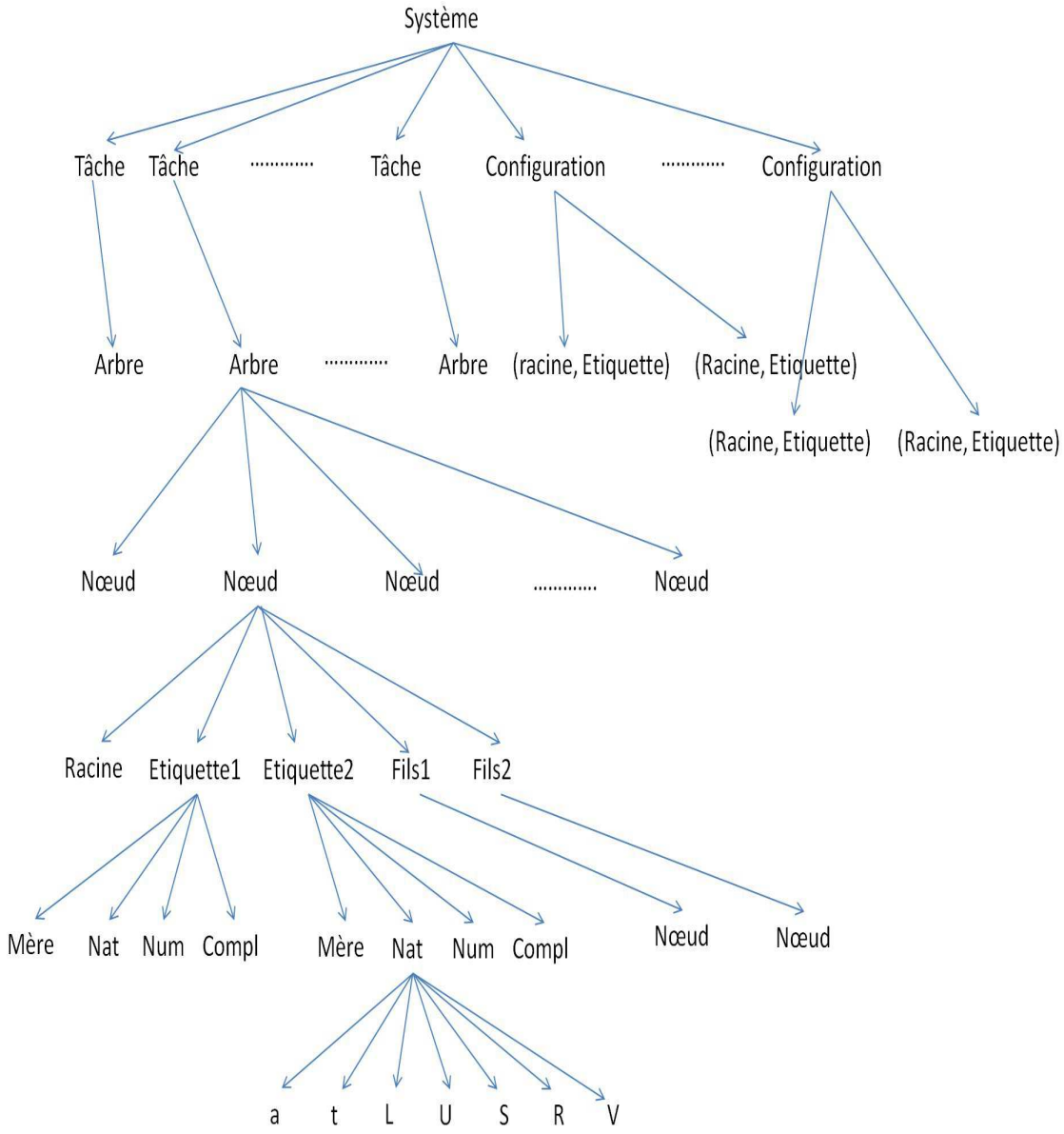


Figure 8.19 – Vue graphique simplifiée du XML Schéma du fichier permettant la construction du modèle de réseau de Petri, approche arbre, des applications

Bilan

Nous avons proposé et validé dans ce chapitre une modélisation, à base de réseaux de Petri, d'applications temps-réel.
Le modèle proposé complète les modèles existants en prenant en compte les relations inter-tâches pouvant exister entre les tâches de l'application.
Nous nous sommes ensuite intéressés à l'obtention de ce modèle, à partir du code de l'application.
L'outil que nous obtiendrons sera complété dans un prochain travail pour la génération des arbres d'ordonnancement.

Chapitre 9

Conclusion générale

"*Chaque publication scientifique ne sert qu'à poser 10, 20 questions. Chaque découverte scientifique est passionnante parce qu'elle ouvre un univers de questions. Si les questions vous angoissent, ne soyez pas scientifique*"
Boris Cyrulnik

Sommaire

Ce dernier chapitre récapitule les résultats obtenus durant notre thèse, puis présente les perspectives qui découlent de ce travail.

9.1 Conclusion

Nous commençons par rappeler quels étaient nos objectifs initiaux, puis nous présentons nos résultats obtenus ainsi que leur importance sur les travaux dans le domaine.

9.1.1 Reprise des objectifs

L'objectif principal de notre travail de thèse était d'intégrer les aspects sémantiques pendant l'analyse d'ordonnançabilité des applications temps-réel.
Concrètement, le travail consistait à s'intéresser, pendant l'étude des applications temps-réel, à l'impact de la prise en compte explicite des instructions conditionnelles pendant les phases de modélisation et de validation.
De façon plus précise, il s'agissait de regarder d'une part, à l'intérieur d'une tâche, les relations qui concernent les corrélations entre les comportements choisis dans des conditionnelles successives, portant sur une même valeur, et d'autre part, entre plusieurs tâches, les relations qui proviennent du fait que si elles examinent le même paramètre pour décider de leur comportement, leurs comportements respectifs seront liés.

Nous les avons appelées relations d'ordre sémantique.
L'un des objectifs de l'étude des relations d'ordre sémantique est la validation du comportement effectif de l'application pendant son exécution.
C'est la raison pour laquelle nous avons choisi les approches de validation hors-ligne, qui consistent à trouver une séquence d'ordonnancement valide de l'application à étudier.

9.1.2 Résultats obtenus

Un premier travail a consisté à faire un état de l'art des approches existantes qui considéraient explicitement les tests conditionnels pendant l'analyse d'ordonnançabilité des applications.
Il en ressort principalement que :

1. notre travail porte sur des systèmes dont les tâches ont plus de propriétés et sont plus générales que les tâches considérées dans d'autres travaux.
   Nous considérons en effet les systèmes avec des tâches dépendantes, qui peuvent échanger des messages, partager des ressources, qui sont à départs simultanés ou différés, qui sont à échéance sur requête ou pas ... ;
2. les relations que nous nous proposons d'étudier n'ont pas encore, à notre connaissance, été étudiées par les auteurs dans la littérature.
   Certaines études vont dans ce sens, mais les objectifs des auteurs dans ces cas ne sont pas la validation explicite du comportement des applications, comme c'est notre cas.
   Par conséquent, les techniques de validation sont le plus souvent en-ligne.

Pour prendre explicitement en compte les instructions conditionnelles pendant la modélisation des tâches, nous avons choisi de considérer une approche arborescente, qui représente de façon détaillée tous les comportements d'une tâche, contrairement aux approches linéaires qui les encapsulent.
Ainsi, les primitives temps-réel sont représentées sur leur branche effective, contrairement à l'approche linéaire où elles sont toutes représentées sur le comportement le plus long, même quand ce n'est pas le cas en réalité.
Mais surtout, nous avons ainsi pu étudier à l'intérieur d'une tâche les relations sémantiques entre les sous comportements successifs d'une tâche portant sur la même valeur d'un test en entrée.
Cela nous a permis pendant notre étude, de ne pas considérer à l'intérieur d'une tâche les comportements incorrects, dû à des choix successifs de tests incompatibles portant sur la même valeur.
Les comportements considérés sont donc des comportements effectifs.
Nous avons enfin retenu deux approches pour l'étude des relations conditionnelles : une approche chemin qui représente une tâche comme un ensemble de comportements (chemins), chaque chemin correspondant à un choix d'exécution de la tâche, et une approche arbre, qui représente une tâche comme un arbre d'exécution.
Ces deux approches de modélisation d'une tâche ont été définies de façon formelle.

Ensuite nous avons considéré l'application dans sa globalité.
Nous avons mis en évidence les relations incompatibles qui peuvent exister entre deux comportements de deux tâches de l'application, ces deux comportements provenant de résultats de tests incompatibles.
Nous avons donc étendu la notion de tests incompatibles à l'intérieur d'une tâche, aux tests incompatibles entre plusieurs tâches.
Il en ressort que la condition pour comparer deux tâches est que les dates de réveil et les périodes de ces tâches soient égales.
Ensuite, nous avons étudié l'ensemble des résultats des tests incompatibles suivant qu'on considère l'approche chemin ou l'approche arbre.
Nous avons ainsi remarqué qu'en utilisant l'approche chemin, il pouvait y avoir des comportements incompatibles redondants, dû à la redondance des résultats de tests incompatibles induisant le même comportement.
De façon concrète, si deux comportements incompatibles sont induits chacun par deux tests incompatibles entre eux, il peut arriver que les tests en amont induisent la relation d'incompatibilité, rendant ainsi les tests en aval redondants.
Pour pallier cela, nous avons proposé et validé un ensemble minimal de résultats de tests incompatibles.
L'ensemble des résultats de tests incompatibles permet d'implémenter l'approche chemin, et l'ensemble minimal est utilisé lorsqu'il s'agit d'implémenter l'approche arbre.
Notons que malgré cet ensemble minimal, l'ensemble des comportements incompatibles peut avoir des doublons, du fait par exemple des résultats de tests incompatibles croisés sur les comportements, qui ne peuvent pas être éliminés pendant la construction de l'ensemble minimal.
Ces doublons seront éliminés de façon manuelle.
Enfin, pour ne pas obtenir pendant l'analyse d'ordonnançabilité uniquement des ordonnancements conservatifs, nous avons considéré la tâche oisive.
Elle permettra d'insérer des temps creux dans les ordonnancements, même s'il y a des tâches prêtes à être exécutées.
Pour la représenter en tenant compte des fluctuations de durée liées aux instructions conditionnelles, nous choisissons d'adopter l'approche de la plus courte durée, qui consiste à considérer la tâche oisive avec sa durée d'exécution minimale, qui correspond au facteur de charge maximal de l'application.
Lorsqu'on prend en compte les relations sémantiques, il peut arriver que U_{max} (le facteur de charge de l'application qui correspond au choix par chaque tâche de l'application du comportement qui a la plus longue durée pendant son exécution) soit plus grand que 1, et que l'application soit malgré cela ordonnançable, car le comportement correspondant à U_{max} contient au moins deux comportements incompatibles.
Cela nous a amenés à distinguer 2 cas pendant la modélisation de la tâche oisive : le cas où $U_{max} \leq 1$ et le cas où $U_{max} > 1$ et $U_{min} < 1$.
La modélisation de la tâche oisive a considéré ces deux cas.

Nous avons donc choisi les approches arborescentes, parce qu'elles sont plus adaptées pour modéliser les instructions conditionnelles.
Ce choix nous a naturellement conduit à revoir l'approche de validation.
Nous n'utilisons plus les séquences d'ordonnancement, qui sont linéaires, surconsomment les ressources et encapsulent les instructions conditionnelles, pour la validation de l'application, mais les arbres d'ordonnancement.
Les arbres d'ordonnancement, tout comme les séquences d'ordonnancement, sont définies sur une fenêtre minimale sur laquelle la validation du système garantit l'absence définitive de faute temporelle.
Ils permettent de représenter de façon détaillée le comportement de l'application, car ils tiennent compte de toutes les instructions conditionnelles.
De plus, il n'y a pas de surconsommation de ressources, car les primitives temps-réel s'exécutent sur les comportements effectifs.
Nous les avons définis de façon formelle, et caractérisé la notion d'arbre valide, qui permettra de conclure qu'une application est valide.
Les arbres que nous proposons sont valides sur les plans sémantique, car ils intègrent les relations d'incompatibilité, structurel, car ils prennent en compte toutes les primitives temps-réel en tenant compte de leur agencement, et temporel.
Nous avons proposé des algorithmes de validation des arbres qui permettent de gérer le partage des ressources et l'échange des messages (évitant ainsi des situations d'interblocage), les relations entre les tâches d'une application induites par des tests en commun (évitant ainsi les contradictions sémantiques), les aspects temporels des tâches pendant leur exécution tels que les dépassements d'échéance.

Ensuite, nous avons étudié l'apport de cette technique de validation arborescente, par rapport aux techniques de validation linéaire.
Nous avons tout d'abord montré qu'en l'absence d'instructions conditionnelles, les deux approches conduisent au même résultat : si une application temps-réel est modélisée et validée suivant l'approche linéaire, si elle est ordonnançable, alors elle est ordonnançable suivant l'approche arborescente et réciproquement.
Ce résultat était prévisible, car en l'absence d'instructions conditionnelles, le modèle linéaire d'une tâche est semblable à son modèle arborescent.
En présence d'instructions conditionnelles, l'approche linéaire est parfois pessimiste.
De façon précise, nous avons montré qu'une application peut être déclarée non ordonnançable suivant l'approche linéaire (donc elle n'admet pas de séquence d'ordonnancement valide), alors qu'elle est ordonnançable suivant l'approche arborescente (donc on peut trouver un arbre d'ordonnancement valide).
Par contre, lorsqu'elle est ordonnançable suivant l'approche linéaire, elle reste ordonnançable en utilisant l'approche arborescente.
Les techniques de validation arborescentes apportent donc un avantage en évitant le sur-dimensionnement du matériel, permettant ainsi de faire des économies.
De plus, les résultats obtenus en utilisant les techniques arborescentes sont les résultats réels : si elles amènent à conclure que l'application n'est pas valide, alors elle n'est vraiment pas valide, contrairement aux approches basées sur les techniques

linéaires.
Enfin, elles apportent un aspect *qualité* dans la modélisation et la validation des systèmes temps-réel.

Nous nous sommes ensuite intéressés à l'obtention des arbres d'ordonnancement que nous avons défini.
En première approche, nous avons construit un générateur d'arbres d'ordonnancement **GENTREE**.
Ce générateur prend en entrée l'ensemble des instances des tâches de l'application, et génère en sortie, sur une profondeur donnée (qui correspond à la profondeur minimale sur laquelle la validation est jugée correcte), l'ensemble des arbres d'ordonnancement.
Il est conçu pour générer tous les arbres d'ordonnancement possibles sémantiquement corrects. C'est à dire des arbres pour lesquelles les relations sémantiques ont déjà été prises en compte.
Par contre, il peut générer des arbres qui ne sont pas valides des points de vue structurels et temporels.
Les algorithmes que nous avons proposés permettent donc de les éliminer pour n'en retenir que les valides.
D'autre part, la méthode de génération proposée est exponentielle en fonction du nombre de tâches en entrée.
En effet, l'approche de génération consiste à construire autant d'arbres que de combinaisons possibles de comportements possibles de toutes les tâches de l'application.
Le générateur n'est donc adapté que pour de très petites applications, avec des tâches en nombre réduit et des périodes faibles.
Enfin, puisqu'il génère de façon *brute* tous les arbres d'ordonnancement, il n'est pas possible de faire le choix de ne générer des arbres particuliers.
Il ne nous permet par exemple pas d'obtenir uniquement des arbres dont le temps de réponse moyen est minimal. Il faut pour cela les générer tous, et les tester un par un.
Toutes ces raisons nous ont amenées à refléchir sur une autre approche d'obtention des arbres d'ordonnancement.

Nous avons choisi une approche orientée modèle, basée sur les réseaux de Petri, qui permettra d'obtenir les arbres voulus, par construction du graphe des marquages et dont la complexité peut être réduite grâce à de bonnes heuristiques.
Une telle approche a déjà été utilisée par [Grolleau 1999] pour la génération des séquences d'ordonnancement et par [Pailler 2006] pour la modélisation des tâches comportant des instructions conditionnelles.
De plus, les réseaux de Petri offrent la possibilité de modéliser l'échange de messages, le partage des ressources, le parallélisme, la concaténation et le choix d'instructions, les blocs pré-emptifs ou non et de prendre en compte le temps.
Par conséquent, les réseaux de Petri nous permettront de modéliser et d'analyser les systèmes temps-réel de façon détaillée, en intégrant les contraintes structurelles,

temporelles et comportementales de l'application.

Nous avons donc choisi les réseaux de Petri pour étudier les applications temps-réel suivant l'approche chemin et l'approche arbre.
L'approche chemin nous a permis de valider notre modèle, mais est redondante, parce qu'elle duplique les blocs généraux qui ne comportent pas de conditionnelles dans chacun des comportements. Elle renvoie à la tâche vue comme un ensemble de comportements et est utilisée pour implémenter l'ensemble des comportements incompatibles.
L'approche arbre, qui n'est pas redondante, renvoie à la notion de la tâche vue comme un arbre d'exécution, et est utilisée pour implémenter l'ensemble minimal des comportements incompatibles.
Dans les deux cas, nous avons utilisé un réseau de Petri avec marquage initial pour prendre en compte les dates de premier réveil des tâches, *P-coloré* (coloré suivant les places) simplifié (car nous utilisons juste deux couleurs) pour modéliser l'activation et la terminaison des tâches, fonctionnant sous la règle du tir maximal pour prendre en compte le temps, avec ensemble terminal pour gérer les contraintes liées aux échéances.
Pour prendre en compte les relations d'ordre sémantique entre les tâches, nous avons étendu le modèle de réseau de Petri de [Pailler 2006].
Nous avons ainsi ajouté aux couches structurelles et temporelles existantes une couche sémantique pour modéliser les relations d'incompatibilité entre les comportements des tâches.
Cela nous a conduit à compléter la couche temporelle afin d'assurer la cohérence sémantique de l'application.
Notre modèle a été validé sur l'approche chemin. L'approche arbre étant une factorisation, notre preuve peut être étendue.
Nous avons donc montré que :

1. nous ne pouvons pas exécuter dans une même période deux chemins incompatibles ;
2. aucun comportement n'est définitivement perdu : à chaque activation, chaque tâche dispose bien de tous ses chemins effectifs.

Ensuite, nous avons étudié la tâche oisive.
Nous avons proposé une modélisation de la tâche oisive prenant en compte les cas $U_{max} \leq 1$ et $U_{max} > 1$.
Ce deuxième cas a nécessité l'utilisation d'un réseau de Petri synchronisé, avec arc inhibiteur pour la réinitialisation de la tâche oisive.
Notons que l'étude de la tâche oisive a uniquement porté sur les tâches à départs simultanés et que la modélisation de la tâche oisive est la même, que l'on considère l'approche arbre ou l'approche chemin.
Ce qui change, c'est son comportement vis à vis des autres tâches de l'application, selon qu'on choisit un chemin (approche chemin), ou bien une branche (approche arbre).

Enfin, nous avons posé les bases (structures de données, méthodologies) pour la construction de **PETRIGEN**, un générateur de modèles de réseau de Petri, approches chemin et arbre, à partir du pseudo-code C d'une application temps-réel. **PETRIGEN** utilise les analyseurs syntaxique et sémantique **FLEX** et **BISON**, pour produire des fichiers XML à partir desquels seront construits les réseaux de Petri.

La figure 9.1 récapitule ce qui a été fait dans cette thèse, et ce qui reste à faire dans l'immédiat.

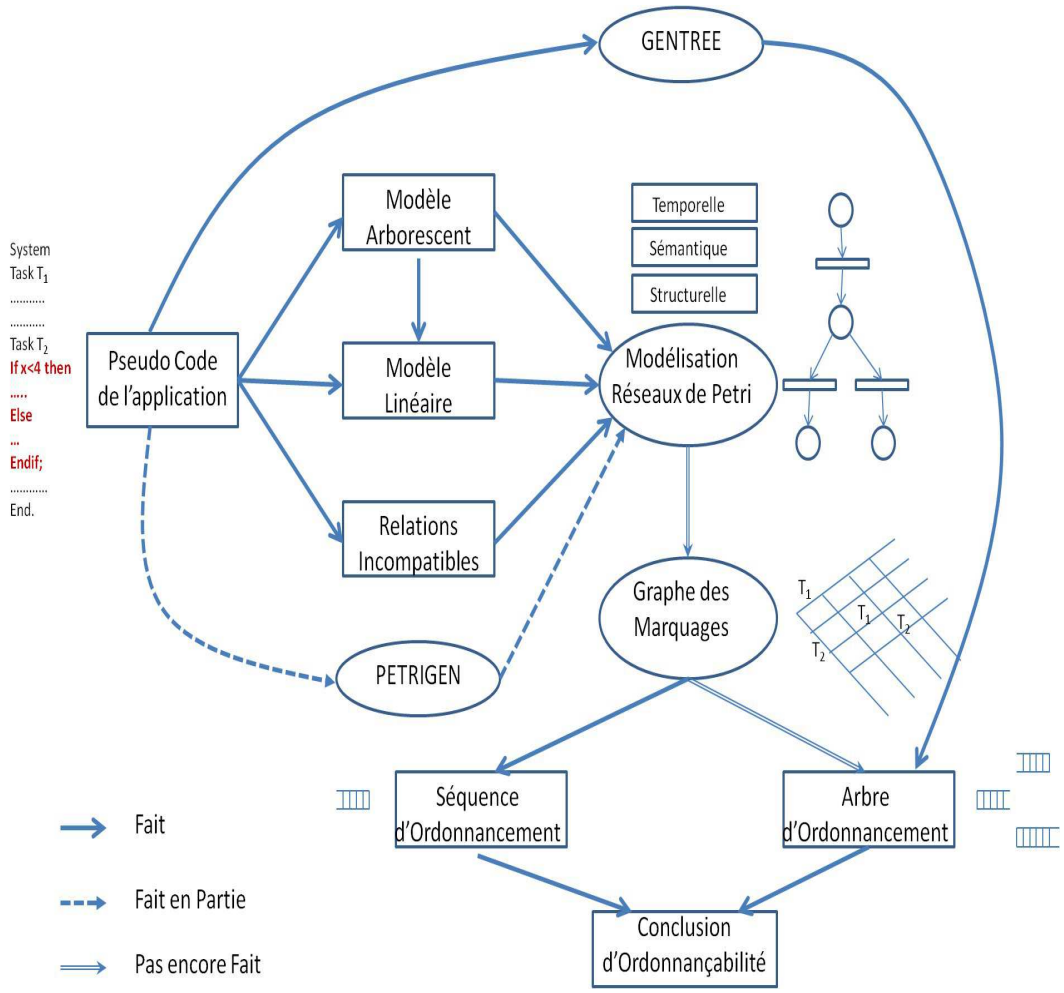


Figure 9.1 – Vue complète de l'étude des applications temps-réel avec prise en compte explicite des instructions conditionnelles et des relations d'ordre sémantique

9.2 Perspectives

Faute de temps, nos objectifs initiaux ne sont pas complètement atteints.
Nous présentons dans cette section les perspectives immédiates, qui concernent les points qui ont été abordés dans ce travail, mais doivent être complétés, puis les perspectives à long terme.

9.2.1 À court terme

En premier lieu, nous allons finaliser la construction de l'outil **PETRIGEN**, qui nous donnera pour un système temps-réel en entrée son modèle de réseau de Petri, que l'on considère l'approche chemin ou l'approche arbre.

Ensuite, nous allons étudier la modélisation de la tâche oisive dans le cas des tâches à départs différés.

Enfin, l'un de nos objectifs va consister à construire les arbres d'ordonnancement à partir de ces modèles, par construction du graphe des marquages.
On pourra ainsi construire uniquement des arbres qualitatifs, et proposer des heuristiques pour réduire la complexité.
Cette perspective a fait l'objet d'une proposition de thèse.

9.2.2 À long terme

Nous réfléchissons sur l'utilisation des arbres d'ordonnancement dans la pratique.
Une thèse est en cours au laboratoire, qui étudie la mise en œuvre des séquences d'ordonnancement.
Les résultats obtenus pourront être étendus aux arbres d'ordonnancement.

Une autre perspective serait l'extension de ce travail aux cas multiprocesseur et distribué, qui sont de plus en plus utilisés par les industries.

Il pourrait aussi être intéressant d'étendre l'ensemble des tests utilisés, pour considérer des systèmes plus complets.
Les tests ne seront plus réduits aux seuls tests de seuillage ou d'identification, mais aux tests logiques *and*, *or* ....

Annexe

"*La science doit s'accommoder à la nature. La nature ne peut s'accommoder à la science*"
Ferdinand Brunot

.1 Structure des tâches des essuie-glaces

Nous donnons figure 2 la structure des tâches des essuie-glaces.

```
Tâche T_Reception
b(2); - - lit la position du comodo (elle peut être stop/1/2/3)
S(M_1); - - envoie les informations sur l'état du comodo
End;

Tâche T_Calcul
Input contact;
R(M_1); - - reçoit les informations sur l'état du comodo
b(2); - - extrait la vitesse du véhicule et la quantité d'eau de la trame
If (contact = 0) [durée du test = 1] then - - si le contact est off
        b(1); - - évalue la commande à transmettre
        S(M_2); - - transmet la commande à la tâche T_Coordonne
Else
        b(3); - - évalue la commande à transmettre
        S(M_2); - - transmet la commande à la tâche T_Coordonne
Endif
End;

Tâche T_Coordonne
Input contact;
R(M_2); - - reçoit l'ordre
b(1); - - récupère l'état du contact du véhicule
If (contact = 0) [durée du test = 1] then - - si le contact est off
        b(2); - - coordonne le balayage des essuies-glaces
Else
        b(1); - - coordonne le balayage des essuies-glaces
End;
```

Figure 2 – Structure des tâches des essuie-glaces

.2 Algorithme de génération des arbres

Nous commençons par définir quelques structures de données nécessaires pour la compréhension de l'algorithme :

Structure de Données .2.1 :étiquette

- - l'étiquette de l'arbre d'une tâche est caractérisée par :
- - le numéro de la tâche mère qui est un entier,
- - la nature (nat) de l'étiquette qui est un caractère :
- - - - a pour une instruction déclarative, t pour un test ;
- - - - L pour une prise de ressource, U pour une libération de ressource ;
- - - - S pour un envoi de message, R pour une réception ;
- - - - et enfin V pour quand aucune instruction n'est exécutée (θ).
- - le numéro (num) de l'étiquette. Ce numéro a plusieurs rôles :
- - - - si nat = L ou U, num = numéro de la ressource ;
- - - - si nat = S ou R, num = numéro du message ;
- - - - si nat = t, num = numéro du test ;
- - - - dans tous les autres cas, num = 0.
- - et enfin un entier complément (compl), qui a plusieurs rôles :
- - - - si nat = t, compl = signe du test ;
- - - - si nat = L ou U ou a ou V, compl = 0 ;
- - - - si nat = S, compl = numéro du destinataire ;
- - - - si nat = R, compl = numéro de la tâche émettrice.

étiquette est un enregistrement
{
    mère : entier ;
    nat : caractère ;- - a, t, L, U, S, R, V
    num : entier ;
    compl : entier ;- - voir commentaires
} étiquette.

Une étiquette de l'arbre d'une tâche est donc un quadruplet de $\mathbb{N}$x$Char$x$\mathbb{N}$x$\mathbb{N}$ qui peut être sous l'une des formes suivantes :

$$\begin{cases} (numtache,\ a,\ 0,\ 0) \\ (numtache,\ t,\ j,\ s)\ j\ numero\ du\ test\ et\ s\ son\ signe (0 \leftrightarrow -, 1 \leftrightarrow +) \\ (numtache,\ L,\ k,\ 0)\ k\ numero\ de\ la\ ressource \\ (numtache,\ U,\ k,\ 0)\ k\ numero\ de\ la\ ressource \\ (numtache,\ S,\ k,\ l)\ k\ numero\ du\ message\ et\ l\ le\ destinataire \\ (numtache,\ R,\ k,\ l)\ k\ numero\ du\ message\ et\ l\ le\ numero\ emetteur \\ (0,\ V,\ 0,\ 0)\ quand\ rien\ ne\ se\ passe \end{cases}$$

On représente un arbre d'ordonnancement comme un arbre binaire étiqueté. Ses nœuds sont caractérisés par la structure de donnée .2.2 :

Structure de Données .2.2 :noeud

- - un noeud de l'arbre d'ordonnancement est caractérisé par :
- - - - sa clé qui est un entier. Cette clé détermine la
- - - - durée d'exécution de toutes les instructions déjà rencontrées.
- - - - deux étiquettes, et1 et et2 :
- - - - - - et1 est utilisée pour la construction de l'arbre quand il n y' a pas
- - - - - - de tests,
- - - - - - ou pour considérer la partie then quand il y a un test conditionnel ;
- - - - - - et2 est utilisée pour prendre en compte la partie else des tests
- - - - deux liens fils1 et fils2 d'étiquettes et1 et et2 :
- - - - chaque lien pointe sur son fils correspondant ;
- - - - - - les liens sont des pointeurs sur les noeuds, ce qui permet de définir
- - - - - - l'arbre récursivement.

noeud est un enregistrement
{
    cle : entier ;
    et1, et2 : étiquette ;
    fils1, fils2 : lien ;
} noeud.

La structure d'un arbre d'ordonnancement est schématiquement représentée figure 3.

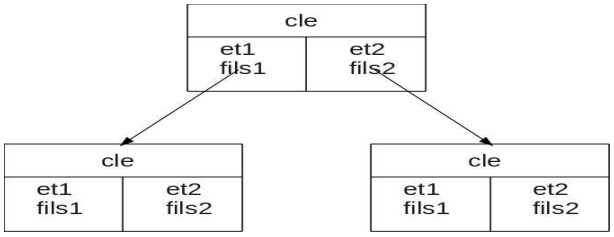


Figure 3 – Une illustration de la structure d'un arbre d'ordonnancement

La structure des tâches manipulées est semblable à celle des arbres. Nous ne la représentons donc pas.
L'unique différence se trouve au niveau du sens de l'enregistrement mère de étiquette.
Dans le cas des arbres de tâche, il s'agit du numéro de la tâche qu'on représente.
Notre générateur d'arbres prend en entrée la liste des instances des tâches activées entre les instants Début et Fin sur lesquels on veut construire les arbres.
Pour des besoins d'implémentation, cette liste est initialement rangée en ordre croissant de date d'activation.
Notons que cette liste d'instances est obtenue par calcul direct sur les tâches données en entrée. La structure de données .2.3 caractérise les instances.

Structure de Données .2.3 :instance

- - une instance d'une tâche est caractérisée par :
- - - - un pointeur sur la racine de l'arbre ;
- - - - le numéro de la tâche dont elle est instance ;
- - - - sa date de réveil ;
- - - - son numero.

instance est un enregistrement
{
rac : pointeur ; - - rac pointe initialement sur la racine de l'arbre, et à un
- - instant donné, sur le sous
- - arbre correspondant à la portion de la tâche restant à exécuter
mère : entier ;
$date\_rev$: entier ;
num_inst : entier ;
} instance.

Ainsi, l'instance $I_{i,i_j}^{d_j}$ est la i^{eme} instance, activée à la date $t = d_j$, de la tâche mère T_{i_j}, et sa racine est la racine de l'arbre d'exécution de la tâche T_{i_j}.
Nous définissons enfin l'ensemble, noté $test\_passe$, des tests déjà rencontrés pendant la génération des arbres d'ordonnancement.
Cet ensemble est initialement vide et se remplit au fur et à mesure que l'arbre est construit. La particularité des tests ajoutés dans cette liste est qu'on leur ajoute un paramètre, le numéro de l'instance d'où provient le test. Ce numéro est important pour la gestion des relations incompatibles.
Un test de l'ensemble $test\_passe$ est donc un élément de la forme
(mère, num, s, $num\_instance$) avec :

1. mère : numéro de la tâche mère ;
2. num : numéro du test ;
3. s : signe du test (0 ou 1) ;
4. $num\_instance$: le numéro de l'instance considérée.

Nous disposons maintenant de toutes les briques pour décrire l'algorithme de génération des arbres :

Algorithme 1: Construit

```
/*Liste des instances dans l'ordre croissant des dates
  d'activation                                              */
input  : Linst ;
/*Tests effectués entre la racine et le noeud courant       */
input  : test_passe ;
/*Un instant quelconque                                     */
input  : t ;
/*La durée de simulation                                    */
input  : Fin
/*La liste des arbres d'ordonnancement                      */
output : L ;
/*si on arrive à la fin de la simulation                    */
si t = Fin alors
    L = TraitementFinSimulation(Linst,test_passe,Fin);
    retourner L;
/*sinon, si il n y'a plus d'instances à traiter             */
sinon si Linst = [] alors
    L = TraitementPlusDinstancesATraiter(Linst,test_passe,t,Fin);
    retourner L;
/*sinon, si il n y'a pas de tâches prêtes à l'instant t     */
sinon si Tete(Linst).date_rev > t alors
    L = TraitementInstancesNonReveille(Linst,test_passe,t,Fin);
    retourner L;
/*sinon, traitement des instances réveillées                */
sinon
    L = TraitementInstancesReveille(Linst,test_passe,t,Fin);
    retourner L;
fin
retourner L;
```

Algorithme 2: TraitementFinSimulation

```
/*Module de traitement quand on est en fin de simulation       */
input  : Linst ;
input  : test_passe ;
input  : Fin
output : L ;
/*garder les instances activées commençant par une PTR         */
LPTR := [];
pour A dans Linst faire
    si A.date_rev ≤ t et A.rac.et1.nat ∈ {L, U, S, R} alors
        LPTR := LPTR + A;
    fin
fin
L := [];
si LPTR ≠ [] alors
    pour T ∈ LPTR faire
        LPTR_aux := Tronque (LPTR, T);
        Laux := Construit (LPTR_aux, test_passe, Fin, Fin);
        L1 := Insere ((T.mère,T.et1.nat,T.et1.num,T.et1.compl),Laux,Fin);
        L := L + L1;
    fin
sinon
    B := Creenoeud (Fin,(0,V,0,0),(0,V,0,0),null,null);
    L := L + B;
fin
retourner L;
```

Algorithme 3: TraitementPlusDinstancesATraiter

```
/*Module de traitement quand il n' y a plus d'instances à traiter
  */
input  : Linst ;
input  : test_passe ;
input  : t ;
input  : Fin
output : L ;
L1 := Construit ([],test_passe,t+1,Fin);
/*on ajoute un temps d'inactivité                                  */
L := Insere ((0,V,0,0),L1,t);
retourner L;
```

Algorithme 4: TraitementInstancesNonReveille

```
/*Module de traitement des instances non réveillées               */
input  : Linst ;
input  : test_passe ;
input  : t ;
input  : Fin
output : L ;
/*pas de tâches prêtes à l'instant t, on ajoute un temps
  d'inactivité                                                    */
L1 := Construit (Linst,test_passe,t+1,Fin);
L := Insere ((0,V,0,0),L1,t);
retourner L;
```

Algorithme 5: TraitementInstanceReveille

```
/*Module de traitement des instances réveillées                    */
input  : Linst ;
input  : test_passe ;
input  : t ;
input  : Fin
output : L ;
/*on partitionne l'ensemble des tâches en fonction de la nature
  de leur première action                                          */
Ltest := [];
Limp := [];
LPTR := [];
Ldort := [];
pour A ∈ Linst faire
    /*on récupère les instances qui ne sont pas activées           */
    si A.date_rev > t alors
        Ldort := Ldort + A;
    /*on récupère les instances qui commencent par une instruction
      impérative                                                   */
    sinon si A.rac.et1.nat = a alors
        Limp := Limp + A;
    /*on récupère les instances qui commencent par une primitive
      temps-réel                                                   */
    sinon si A.rac.et1.nat ∈ {L, U, S, R} alors
        LPTR := LPTR + A;
    /*on récupère les instances qui commencent par un test         */
    sinon
        Ltest := Ltest + A;
    fin
fin
L := [];
/*on traite les tâches commençant par des instructions
  déclaratives                                                     */
TraitementInstancesDeclaratives();
/*cas des primitives temps-réel                                    */
TraitementInstancesPTR();
/*cas des tests                                                    */
TraitementInstancesTests();
retourner L;
```

Algorithme 6: TraitementInstancesDeclaratives

```
/*Module de traitement des instances déclaratives                    */
pour T ∈ Limp faire
    Linst_aux := Ldort + LPTR + Ltest + Tronque (Limp,T);
    Laux := Construit (Linst_aux,test_passe,t+1,Fin);
    /*on commence par l'instruction déclarative                      */
    L1 := Insere ((T.mère,a,0,0),Laux,t);
    L := L + L1;
fin
```

Algorithme 7: TraitementInstancesPTR

```
/*Module de traitement des instances commençant par des PTR          */
pour T ∈ LPTR faire
    Linst_aux := Ldort + Limp + Ltest + Tronque (LPTR,T);
    /*la primitive étant de durée nulle, on n'incrémente pas le
      temps                                                          */
    Laux := Construit (Linst_aux,test_passe,t,Fin);
    /*on commence par une PTR                                        */
    L2 := Insere ((T.mère,T.et1.nat,T.et1.num,T.et1.compl),Laux,t);
    L := L + L2;
fin
```

Algorithme 8: TraitementInstancesTests

```
/*Module de traitement des instances commençant par un test          */
pour T ∈ Ltest faire
    /*on récupère le test correspondant à la branche then (+)        */
    te1 := T.rac.et1;
    /*on récupère le test correspondant à la branche else (-)        */
    te2 := T.rac.et2;
    /*on initialise les variables booléennes relatives aux tests à
      faux. Ceci suppose qu'elle ne sont pas incompatibles avec
      les tests déjà rencontrés                                      */
    paste1 := false;
    paste2 := false;
    TraitementInstancesTestsSuite();
fin
```

Algorithme 9: TraitementInstancesTestsSuite

```
/*Suite du module de traitement des instances commençant par un
  test                                                          */
/*on regarde chacun des tests, pour voir s'il n y' a pas
  incompatibilité avec un test préalable                        */
/*cas du test te1                                               */
pour te ∈ test_passe faire
    si T.num_instance = te.num_instance alors
        si [(te1.mère ≤ te.mère) et (Incompatible (te1,te))] ou [(te.mère <
        te1.mère) et (Incompatible (te,te1))] alors
            paste1 := true;
        fin
    fin
fin
/*cas du test te2                                               */
/*faire le même traitement que précédemment                     */
/*on ressort donc de ces traitements en sachant si chacun des
  tests récupérés est incompatible avec un test déjá rencontré */
/*si les deux tests sont incompatibles avec les tests déjà
  rencontrés                                                    */
si paste1 et paste2 alors
    /*on ne fait rien, car les deux branches sont incompatibles */
L3 := [];
/*sinon, si seule la branche then est incompatible. On garde le
  test te2                                                      */
sinon si paste1 alors
    L = GarderTestte2(Ldort,Limp,LPTR,Ltest,T);
    retourner L;
/*sinon si seule la branche else est incompatible, on garde le
  test te1                                                      */
sinon si paste2 alors
    L = GarderTestte1(Ldort,Limp,LPTR,Ltest,T);
    retourner L;
/*sinon si les deux branches sont compatibles, on garde les deux
  tests te1 et te2                                              */
sinon
    L = GarderTestte1te2(Ldort,Limp,LPTR,Ltest,T);
    retourner L;
fin
```

Algorithme 10: GarderTestte2

```
/*Module de traitement pour la branche te2                    */
input  : Ldort ;
input  : Limp ;
input  : LPTR ;
input  : Ltest ;
input  : T ;
input  : te2 ;
output : L ;
Linst_aux := Ldort + Limp + LPTR + Tronque 2(Ltest,T);
test_passe_aux := Add (test_passe,te2,T.num_instance);
Laux := Construit (Linst_aux,test_passe_aux,t+1,Fin);
L3 := Insere 2((T.mère,t,te2.num,te2.s),Laux,t);
L := L + L3;
retourner L;
```

Algorithme 11: GarderTestte1

```
/*Module de traitement pour la branche te1                    */
input  : Ldort ;
input  : Limp ;
input  : LPTR ;
input  : Ltest ;
input  : T ;
input  : te1 ;
output : L ;
Linst_aux := Ldort + Limp + LPTR + Tronque (Ltest,T);
test_passe_aux := Add (test_passe,te1,T.num_instance);
Laux := Construit (Linst_aux,test_passe_aux,t+1,Fin);
L3 := Insere ((T.mère,t,te1.num,te1.s),Laux,t);
L := L + L3;
retourner L;
```

Algorithme 12: GarderTestte1te2

```
/*Module de traitement pour les branches te1 et te2                    */
input  : Ldort ;
input  : Limp ;
input  : LPTR ;
input  : Ltest ;
input  : T ;
input  : te1 ;
input  : te2 ;
output : L ;
Linst_aux1 := Ldort + Limp + LPTR + Tronque (Ltest,T);
Linst_aux2 := Ldort + Limp + LPTR + Tronque 2(Ltest,T);
test_passe_aux1 := Add (test_passe,te1,T.num_instance);
test_passe_aux2 := Add (test_passe,te2,T.num_instance);
Laux1 := Construit (Linst_aux1,test_passe_aux1,t+1,Fin);
Laux2 := Construit (Linst_aux2,test_passe_aux2,t+1,Fin);
L3 := Insere 12(T.mère,te1.num,Laux1,Laux2,t);
L := L + L3;
retourner L;
```

Il reste maintenant à définir les fonctions utilisées dans l'algorithme de la fonction Constructeur.

Algorithme 13: Insere

```
/*Cette fonction insère à un instant t un noeud en racine de tous
  les arbres de L et étiquette le lien1 (celui correspondant à
  et1) par lien                                                  */
input  : lien ;
input  : L ;
input  : t ;
/*elle retourne une liste d'arbres actualisés                    */
output : Rep ;
Rep := [];
pour A ∈ L faire
    B := Creenoeud (t,lien,(0,V,0,0),A,null));
    Rep := Rep + B;
fin
retourner Rep;
```

Algorithme 14: Tronque

```
/*Cette fonction prend en entrée une liste d'instances et une
  instance, et retourne cette liste d'instances en tronquant
  l'instance considérée de son premier élément.                  */
/*quand l'instance considérée est un test, l'élément tronqué est
  celui correspondant à la branche then                          */
/*les instances de cette liste d'instances peuvent commencer par
  une instruction déclarative, par une primitive temps-réel ou
  par un test                                                    */
input  : LI ;
input  : T ;
output : LIaux ;
/*on récupère l'instance tronquée                                */
T' := (T.fils1,T.mère,T.date_rev,T.num_instance);
si T'.rac = null alors
    /*s'il n y'a plus d'instructions, on l'élimine              */
    LIaux := LI \ T;
sinon
    /*sinon on tronque en remplaçant T par T'                  */
    /*Pour préserver l'ordre (en fonction des dates de réveil),
      l'insertion se fait "en place"                           */
    LIaux := (LI \ T) + T';
fin
retourner LIaux;
```

Algorithme 15: Tronque2

```
/*Cette fonction prend en entrée une liste d'instances commençant
  par un test et une instance, et retourne cette liste
  d'instances en tronquant l'instance considérée de son premier
  élément.                                                    */
/*la principale différence avec la fonction Tronque est que
  l'élément tronqué est celui correspondant à la branche else */
input  : LT ;
input  : T ;
output : LIaux ;
T' := (T.fils2,T.mère,T.date_rev,T.num_instance);
si T'.rac = null alors
    /*s'il n y'a plus d'instructions, on l'élimine             */
    LIaux := LT \ T;
sinon
    /*sinon on tronque en remplaçant T par T'                 */
    /*Pour préserver l'ordre (en fonction des dates de réveil),
      l'insertion se fait "en place"                          */
    LIaux := (LT \ T) + T';
fin
retourner LIaux;
```

Algorithme 16: Add

```
/*Cette fonction prend en entrée la liste des tests déjà
  rencontrés, un test (celui sur lequel on se trouve) et un
  numéro d'instance (l'instance considérée) et ajoute le test à
  la liste des tests passés                                   */
input  : test_passe ;
input  : te ;
input  : num_instance ;
output : Ltest ;
teaux := (te.mère,te.num,te.s,num_instance);
Ltest := test_passe + teaux;
retourner Ltest;
```

Algorithme 17: Insere2

```
/*La principale différence avec la fonction Insere est que
  l'insertion se fait ici dans le fils2                          */
input  : lien ;
input  : L ;
input  : t ;
output : Rep ;
Rep := [];
pour A ∈ L faire
    B := Creenoeud (t,(0,V,0,0),lien,null,A);
    Rep := Rep + B;
fin
retourner Rep;
```

Algorithme 18: Insere12

```
/*La principale différence avec les fonctions Insere et Insere2
  est que l'insertion se fait ici dans les deux fils fils1 et
  fils2                                                          */
input  : mère ;
input  : num ;
input  : t ;
/*L1 est la liste d'arbres correspondante à la branche then     */
input  : L1 ;
/*L2 est la liste d'arbres correspondante à la branche then     */
input  : L2 ;
output : Rep ;
Rep := [];
pour A1 ∈ L1 faire
    pour A2 ∈ L2 faire
        B := Creenoeud (t,(mère,t,num,1),(mère,t,num,0),A1,A2);
        Rep := Rep + B;
    fin
fin
retourner Rep;
```

Algorithme 19: Incompatible

```
/*Cette fonction retourne vrai quand les tests passés en entrée
  sont incompatibles, et faux sinon                              */
/*les deux tests en entrée                                       */
input  : te1,te2 ;
output : Result ;
Result := te1^{s1}_{num1} ⇒ ¬te2^{s2}_{num2};
retourner Result;
```

.3 Grammaire BNF des systèmes considérés dans notre étude

```
-- Cette grammaire spécifie les tâches que nous manipulons
-- Les mots en gras sont des mots clés
-- Un système de tâches a un nom, éventuellement une liste de ressources, de messages, de variables et au moins
- - deux tâches
UnSysteme: Système nom ListeRess ListeMess ListeVar ListeTach end . ;
-- Il peut ne pas avoir de ressources
ListeRess: { } | Ressources;
Ressources: Ressource ListeNoms;
-- Il peut ne pas avoir de messages
ListeMess: { } | Messages;
Messages: Message ListeNoms;
-- Il peut ne pas avoir de variables
ListeVar: { } | Variables;
Variables: Variable ListeNoms;
-- Il y a au moins deux tâches
ListeTach: UneTache UneTache | UneTache ListeTach;
-- Une tâche est caractérisée par un nom, une date de réveil, un délai critique, une période, une liste de variables
- - (qui peuvent être des variables simples ou des variables Input), et une liste d'instructions
UneTache: Tâche nom nombre nombre nombre begin ListeVariablesSimples ListeVariablesInput
ListesInstructions end ; ;
-- Il peut ne pas avoir de variables simples
ListeVariablesSimples: { } | VariablesSimples;
VariablesSimples: Variable ListeNoms;
-- Il peut ne pas avoir de variables Input
ListeVariablesInput: { } | VariablesInput;
VariablesInput: Input ListeNoms;
ListeNoms: nom | nom ListeNoms;
ListesInstructions: Instruction ; | Instruction ; ListesInstructions;
-- Une instruction peut être un bloc simple, une primitive temps-réel, un bloc conditionnel
Instruction: b ( nombre ) | L ( nom ) | U ( nom ) | S ( nom ) | R ( nom )
-- Les tests ont une durée qui est un nombre
            if ( Test ) [ durée = nombre] then ListesInstructions;
          | if ( Test ) [ durée = nombre] then ListesInstructions else ListesInstructions;
-- Les tests sont de type seuillage ou identification
Test: nom Operateur Valeur;
Valeur: nom | nombre;
Operateur: < | ≤ | > | ≥ | = | ≠ ;
```

Figure 4 – Grammaire **BNF** des systèmes manipulés

.4 XML Schéma des fichiers XML des applications dans l'approche chemin

```
    /* Version 1.0 */
/* Christian Fotsing */
/* xml schema des fichiers xml pour le réseau de Petri approche chemin
*/
< ?xml version="1.0" encoding="utf-8" ?>
<xsd :schema xmlns :xsd="http ://www.w3.org/2001/XMLSchema">
/* Un système a un nom, peut avoir des ressources, des variables, et
envoyer ou recevoir des messages */
/* Un système a une liste de tâches (sous forme de chemins), et une liste
de configurations impossibles (comportements incompatibles) */
/* Il y a autant de configurations impossibles que d'éléments de RIT */
<xsd : element name = "systeme">
    <xsd : Complextype>
    <xsd : sequence>
    <xsd : element name = "ressource" type = "xsd :string" minoccurs = "0"
    maxoccurs = "unbounded"/ >
    <xsd : element name = "message" type = "xsd :string" minoccurs = "0"
    maxoccurs = "unbounded"/ >
    <xsd : element name = "variable" type = "xsd :string" minoccurs = "0"
    maxoccurs = "unbounded"/ >
    <xsd : element name = "tache" type = "tachetype" minoccurs = "2"
    maxoccurs = "unbounded"/ >
    <xsd : element name = "configuration" type = "configurationtype"
    minoccurs = "0" maxoccurs = "unbounded"/ >
    < /xsd : sequence>
    <xsd : attribute name = "nom" use = "required" type = "xsd : string"/ >
    < /xsd : Complextype>
< /xsd : element>
/* Une tâche a éventuellement des variables, et un ensemble de chemins
(comportements) */
/* Une tâche a un nom, une date de réveil, un délai critique et une période
*/
<xsd : Complextype name = "tachetype">
    <xsd : sequence>
    <xsd : element name = "variable" type = "xsd :string" minoccurs = "0"
    maxoccurs = "unbounded"/ >
    <xsd : element name = "chemin" type = "chemintype" minoccurs = "1"
    maxoccurs = "unbounded"/ >
    < /xsd : sequence>
    <xsd : attribute name = "nom" use = "required" type = "xsd : string"/ >
```

```
    <xsd : attribute name = "datereveil" use = "required" type = "xsd : integer"/ >
    <xsd : attribute name = "delaicritique" use = "required" type = "xsd : integer"/ >
    <xsd : attribute name = "periode" use = "required" type = "xsd : integer"/ >
< /xsd : Complextype>
/* Un chemin est une suite d'instructions. Il a un identifiant qui va le
caractériser de façon unique, un nom et sa durée d'exécution */
<xsd : complextype name ="chemintype">
    <xsd : element name = "instruction" type = "instructiuontype"
    minoccurs = "1" maxoccurs = "unbounded"/ >
    <xsd : attribute name = "numero" use = "required" type = "xsd : integer"/ >
    <xsd : attribute name = "dureeexecution" use = "required"
    type = "xsd : integer"/ >
< /xsd : complextype>
/* Une instruction peut être une primitive temps-réel, ou un test */
<xsd : complextype name ="instructiontype">
    <xsd : choice>
    <xsd : element name = "block" type = "blocktype"/ >
    <xsd : element name = "lock" type = "locktype"/ >
    <xsd : element name = "unlock" type = "unlocktype"/ >
    <xsd : element name = "send" type = "sendtype"/ >
    <xsd : element name = "receive" type = "receivetype"/ >
    <xsd : element name = "test" type = "testtype"/ >
    < /xsd : choice>
< /xsd : complextype>
/* Un bloc a une durée */
<xsd : simpletype name ="blocktype">
    <xsd : attribute name = "param" use = "required" type = "xsd : integer"/ >
< /xsd : simpletype >
/* Une ressource peut être prise */
<xsd : simpletype name ="locktype">
    <xsd : attribute name = "param" use = "required" type = "xsd : string"/ >
< /xsd : simpletype >
/* Une ressource peut être libérée */
<xsd : simpletype name ="unlocktype">
    <xsd : attribute name = "param" use = "required" type = "xsd : string"/ >
< /xsd : simpletype >
/* Un message peut être envoyé */
<xsd : simpletype name ="sendtype">
    <xsd : attribute name = "param" use = "required" type = "xsd : string"/ >
< /xsd : simpletype >
/* Un message peut être reçu */
<xsd : simpletype name ="receivetype">
    <xsd : attribute name = "param" use = "required" type = "xsd : string"/ >
< /xsd : simpletype >
```

/* **Un test a une durée est composé d'une variable, d'un opérateur et de la valeur de la variable** */
<xsd : complextype name ="testtype">
<xsd : sequence>
<xsd : attribute name = "param" use = "required" type = "xsd : integer"/ >
<xsd : element name = "variable" type = "xsd : string"/ >
<xsd : element name = "operateur" type = "operateurtype"/ >
<xsd : element name = "valeur" type = "xsd : valeurtype"/ >
< /xsd : sequence>
< /xsd : complextype>
/* **La valeur du test dépend du type de la variable. Il peut s'agir d'un réel ou d'un texte** */
<xsd : simpletype name ="valeurtype">
<xsd : choice>
<xsd : element name = "constante" type = "xsd : real"/ >
<xsd : element name = "texte" type = "xsd : string"/ >
<xsd : choice>
< /xsd : simpletype >
/* **L'opérateur de test peut être** $<$, $\leq$, $>$, $\geq$, $=$ **ou** $\neq$ */
<xsd : simpletype name ="operateurtype">
<xsd : restriction base = "xsd : string">
<xsd : enumeration value = "<" / >
<xsd : enumeration value = ">" / >
<xsd : enumeration value = "≤" / >
<xsd : enumeration value = "≥" / >
<xsd : enumeration value = "=" / >
<xsd : enumeration value = "≠" / >
< /xsd : restriction>
< /xsd : simpletype >
/* **Une configuration de chemins incompatibles est composée d'une liste de deux tâches.**
Chaque tâche est caractérisée par son numéro unique, et le chemin (en fait le numéro unique du chemin) considéré qui intervient dans la relation d'incompatibilité. */
<xsd : Complextype name = "configurationtype">
<xsd : sequence>
<xsd : element name = "config" type = "configtype" minoccurs = "0"
maxoccurs = "unbounded"/ >
< /xsd : sequence>
< /xsd : Complextype>
<xsd : Complextype name = "configtype">
<xsd : sequence>
<xsd : element name = "element" type = "elementtype" minoccurs = "2"
maxoccurs = "2"/ >

```
    < /xsd : sequence>
< /xsd : Complextype>
<xsd : simpletype name ="elementtype">
    <xsd : attribute name = "numtache" use = "required" type = "xsd : integer"/ >
    <xsd : attribute name = "numchemin" use = "required" type = "xsd : integer"/ >
< /xsd : simpletype >
```

.5 XML Schéma des fichiers XML des applications dans l'approche arbre

```
    /* Version 1.0 */
/* Christian Fotsing */
/* xml schema des fichiers xml pour le réseau de Petri approche arbre */
< ?xml version="1.0" encoding="utf-8" ?>
<xsd :schema xmlns :xsd="http ://www.w3.org/2001/XMLSchema">
/* Un système a un nom, peut avoir des ressources, des variables, et envoyer ou recevoir des messages */
/* Un système a une liste de tâches (sous forme d'arbre), et une liste de configurations impossibles (sous comportements incompatibles) */
/* Il y a autant de configurations impossibles que d'éléments de RIT_min */
<xsd : element name = "systeme">
    <xsd : Complextype>
    <xsd : sequence>
    <xsd : element name = "ressource" type = "xsd :string" minoccurs = "0"
    maxoccurs = "unbounded"/ >
    <xsd : element name = "message" type = "xsd :string" minoccurs = "0"
    maxoccurs = "unbounded"/ >
    <xsd : element name = "variable" type = "xsd :string" minoccurs = "0"
    maxoccurs = "unbounded"/ >
    <xsd : element name = "tache" type = "tachetype" minoccurs = "2"
    maxoccurs = "unbounded"/ >
    <xsd : element name = "configuration" type = "configurationtype"
    minoccurs = "0" maxoccurs = "unbounded"/ >
    < /xsd : sequence>
    <xsd : attribute name = "nom" use = "required" type = "xsd : string"/ >
    < /xsd : Complextype>
< /xsd : element>
/* Une tâche a éventuellement des variables, et est représentée par un arbre */
/* Une tâche a un nom, une date de réveil, un délai critique et une période */
<xsd : Complextype name = "tachetype">
    <xsd : sequence>
    <xsd : element name = "variable" type = "xsd :string" minoccurs = "0"
    maxoccurs = "unbounded"/ >
    <xsd : element name = "arbre" type = "arbretype" minoccurs = "1"
    maxoccurs = "1"/ >
    < /xsd : sequence>
    <xsd : attribute name = "nom" use = "required" type = "xsd : string"/ >
```

<xsd : attribute name = "datereveil" use = "required" type = "xsd : integer"/ >
<xsd : attribute name = "delaicritique" use = "required" type = "xsd : integer"/ >
<xsd : attribute name = "periode" use = "required" type = "xsd : integer"/ >
< /xsd : Complextype>
/* **Un arbre est une suite de nœuds. Il a un identifiant (pour dire qu'il s'agit de l'arbre d'une tâche précise) qui va le caractériser de façon unique** */
<xsd : complextype name ="arbretype">
<xsd : element name = "noeud" type = "noeudtype" minoccurs = "1"
maxoccurs = "unbounded"/ >
<xsd : attribute name = "numero" use = "required" type = "xsd : integer"/ >
< /xsd : complextype>
/* **Un nœud est caractérisé par sa racine (qui correspond à la durée d'exécution de la tâche depuis son activation), qui est identifiée de façon unique par un numéro, et composé d'au moins une étiquette et d'un fils (voir annexe .2).** */
<xsd : Complextype name = "noeudtype">
<xsd : sequence>
<xsd : element name = "etiquette" type = "etiquettetype" minoccurs = "1"
maxoccurs = "2"/ >
<xsd : element name = "fils" type = "noeudtype" minoccurs = "1"
maxoccurs = "2"/ >
< /xsd : sequence>
<xsd : attribute name = "racine" use = "required" type = "xsd : integer"/ >
<xsd : attribute name = "numracine" use = "required" type = "xsd : integer"/ >
< /xsd : Complextype>
/* **Une étiquette est composée du numéro de la tâche, de la nature de l'étiquette, du numéro de l'étiquette et de l'entier complément (voir annexe .2).** */
/* **Nous lui associons un numéro unique pour la gestion des configurations impossibles.** */
<xsd : complextype name ="etiquettetype">
<xsd : sequence>
<xsd : element name = "mere" type = "xsd : integer"/ >
<xsd : element name = "nat" type = "nattype"/ >
<xsd : element name = "num" type = "xsd : integer"/ >
<xsd : element name = "compl" type = "xsd : integer"/ >
< /xsd : sequence>
<xsd : attribute name = "numeti" use = "required" type = "xsd : integer"/ >
< /xsd : complextype>
/* **La nature de l'étiquette peut être** a, t, L, U, S, R **ou** V */
<xsd : simpletype name ="etiquettetype">
<xsd : restriction base = "xsd : string">
<xsd : enumeration value = "a" / >

<xsd : enumeration value = "t" / >
<xsd : enumeration value = "L" / >
<xsd : enumeration value = "U" / >
<xsd : enumeration value = "S" / >
<xsd : enumeration value = "R" / >
<xsd : enumeration value = "V" / >
< /xsd : restriction>
< /xsd : simpletype >
/* **Une configuration de sous comportements incompatibles est composée d'une liste de deux tâches.**
Chaque tâche est caractérisée par le numéro unique de la racine de l'arbre de la tâche, et le numéro unique de l'étiquette qui induit le sous comportement (ou plus généralement le sous arbre) qui intervient dans la relation d'incompatibilité. */
<xsd : Complextype name = "configurationtype">
<xsd : sequence>
<xsd : element name = "config" type = "configtype" minoccurs = "0"
maxoccurs = "unbounded"/ >
< /xsd : sequence>
< /xsd : Complextype>
<xsd : Complextype name = "configtype">
<xsd : sequence>
<xsd : element name = "element" type = "elementtype" minoccurs = "2"
maxoccurs = "2"/ >
< /xsd : sequence>
< /xsd : Complextype>
<xsd : simpletype name ="elementtype">
<xsd : attribute name = "numracine" use = "required" type = "xsd : integer"/ >
<xsd : attribute name = "numeti" use = "required" type = "xsd : integer"/ >
< /xsd : simpletype >

Bibliographie liée à la thèse

"*La science ne connait qu'une loi : la contribution scientifique*"
Bertolt Brecht

Rapports internes

1. Christian Fotsing, Annie Geniet, Guy Vidal-Naquet, "*Modélisation de la Prise en compte de la Sémantique dans les Applications Temps-Réel*", LISI, ENSMA, Juin 2010 ;
2. Christian Fotsing, Annie Geniet, Guy Vidal-Naquet, "*Définition Formelle et Construction des Arbres d'Ordonnancement*", LISI, ENSMA, Janvier 2011.

Séminaires

1. Christian Fotsing, "*Modélisation de la Sémantique dans les Applications Temps-Réel*", MEFOSYLOMA, Paris, France, Mai 2010 ;
2. Christian Fotsing, "*Impacts Sociétaux de mon travail de Thèse*", LA ROCHELLE, France, Octobre 2010.

Workshops internationnaux

1. Christian Fotsing, Annie Geniet, Guy Vidal-Naquet, "*A Realistic Model of Real-Time Systems for Efficient Scheduling*", Software Engineering Workshop, Annual IEEE/NASA Goddard, pp. 3-12, 2009 33rd Annual IEEE Software Engineering Workshop, Skôvde, Suède, 2009 ;

Conférences internationales

1. Christian Fotsing, Annie Geniet, Guy Vidal-Naquet, "*Using Semantic Properties for Real-Time Scheduling*", MDD4DRES, Work In Progress Session, Aussois, France, 2009 ;
2. Christian Fotsing, Annie Geniet, Guy Vidal-Naquet, "*Tree Scheduling Versus Sequential Scheduling*", CARS@EDCC, ACM Digital Library with the ISBN 978-1-60558-915-2, Valence, Espagne, Avril 2010 ;
3. Christian Fotsing, Annie Geniet, Guy Vidal-Naquet, "*Modélisation de la Prise en Compte de la Sémantique dans les Applications Temps-Réel : Concepts et Outils*", AFADL2010, Poitiers, France, Juin 2010 ;

Comité de programme

1. JRWRTC'10 : 4^{th} Junior Researcher Workshop on Real-Time Computing, Toulouse, France, Novembre 2010 ;
2. ETR'11, École d'Été Temps-Réel, Session Doctorants, Brest, France, Septembre 2011 ;
3. JRWRTC'11 : 5^{th} Junior Researcher Workshop on Real-Time Computing, Nantes, France, Septembre 2011.

Encadrements

1. Projet **GENTREE** : encadrement officieux d'*Hamet Tamdia*, étudiant actuellement en Master 2 à l'université de Poitiers, pour le développement du projet.

Bibliographie

[Aga] *L'outil Agatha.* Présentation sur le site web. www-list.cea.fr/labos/fr/LLSP/oasis/OASIS-presentation. (Cité en page 25.)

[Aloysius 1996] K. Aloysius, K. Mok et D. Chen. *A multiframe model for real-time tasks.* In in proceedings of the 17^{th} Real-Time Systems Symposium, Washington DC, 1996. IEEE Computer Society Press. (Cité en page 32.)

[Audsley 1993] N. Audsley, A. Burns, M. Richardson, K. Tindell et A.J. Wellings. *Applying New Scheduling Theory to Static Priority Preemptive Scheduling.* Software Engineering Journal, September 1993. (Cité en page 6.)

[Aussaguès 1998] C. Aussaguès et V. David. *A method and a technique to model and ensure timeless in safety critical real-time systems.* In International Conference on Engineering of Complex Computer Systems (ICECCS'98), 1998. (Cité en page 36.)

[Aussaguès 2002] C. Aussaguès, V. David et J. Delcoigne. *Une approche muti-tâche déterministe pour les systèmes embarqués sûrs de fonctionnement.* In LTRE-2002, Janvier 2002. (Cité en page 36.)

[Baker 1974] K.R. Baker et Z.S. Su. *Sequencing with due-dates and early start times to minimize maximum tardiness.* Naval Research Logistic Quartely, vol. 21, pages 171–176, 1974. (Cité en page 47.)

[Baruah 1990] S. Baruah, L. Rosier et R. Howell. *Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor.* Real-Time Systems, vol. 2, pages 301–324, 1990. (Cité en page 47.)

[Baruah 1999] S. Baruah, D. Chen, S. Gorinsky et A. Mok. *Generalized Multiframe Tasks.* Real-Time Systems, vol. 17, no. 1, pages 5–22, July 1999. (Cité en page 32.)

[Baruah 2003] S. Baruah. *Dynamic and Static Priority Scheduling of Recurring Real-time Tasks.* Real-Time Systems, pages 93–128, 2003. Kluwer Academic Publishers, Manufactured in the Netherlands. (Cité en pages 32 et 34.)

[Beauvais 1996] J.P. Beauvais. *Étude d'algorithmes de placement de tâches temps-réel périodiques complexes dans un système réparti.* PhD thesis, École Centrale de Nantes, 1996. 182 p. (Cité en page 47.)

[Beauvais 1998] J.P. Beauvais et A.M. Eplanche. *Affectation de tâches dans un système temps-réel réparti.* Technique et Sciences Informatiques, vol. 17, no. 1, 1998. (Cité en page 47.)

[Bernat 2002] G. Bernat, A. Colin et S. Petters. *WCET analysis of probabilistic hard real-time systems.* In Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS02), Austin, Texas, December 2002. (Cité en page 7.)

[Best 1987] E. Best et C. Fernandez. Notations and terminology on petri net theory. Arbeitspapierer der GMD 195, March 1987. (Cité en page 42.)

[Bratley 1975] P. Bratley, M. Florian et P. Robillard. *Scheduling with earliest start and due date constraints on multiple machines.* Naval Research Logistic Quartely, vol. 22, no. 1, pages 165–173, 1975. (Cité en page 47.)

[Burns 2003] A. Burns, G. Bernat et I. Broster. *A probabilistic framework for schedulability analysis.* In Third International Embedded Software Conference (EMSOFT03), pages 1–15, 2003. (Cité en page 31.)

[Buttazo 1997] G.C. Buttazo. Hard real time computing systems : predictable scheduling algorithms and applications. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park Norwell, MA 02061, USA, 1997. (Cité en pages 9, 34 et 133.)

[Cartensen 1984] H. Cartensen et R. Valk. *Infinite behaviour and fairness in Petri net.* In Advances in Petri nets 188, pages 83–100, 1984. (Cité en page 44.)

[Cavalcante 1997] S.V. Cavalcante. *A Hardware-software co-design system for embedded real-time applications.* PhD thesis, University of Newcastle, July 1997. (Cité en page 47.)

[Choquet-Geniet 1993] A. Choquet-Geniet et G. Vidal-Naquet. Réseaux de petri et systèmes parallèles. Armand Colin, 1993. (Cité en page 42.)

[Choquet-Geniet 2000] A. Choquet-Geniet, E. Grolleau et F. Cottet. *Etude hors ligne d'une application temps réel à contraintes strictes.* Techniques et Sciences Informatiques (TSI), vol. 19, no. 10, pages 1373–1398, 2000. (Cité en pages 42, 47 et 48.)

[Choquet-Geniet 2003] A. Choquet-Geniet. *Panorama de l'ordonnancement temps-réel monoprocesseur.* In École d'été temps-réel, 2003. (Cité en page 9.)

[Choquet-Geniet 2004] A. Choquet-Geniet et E. Grolleau. *Minimal Schedulability Interval for Real-Time Systems of Periodic Tasks with Offsets.* Theoretical of Computer Sciences, vol. 310, no. 10, pages 117–134, 2004. (Cité en pages 9, 104, 110 et 111.)

[Choquet-Geniet 2006] A. Choquet-Geniet. Les réseaux de petri, un outil de modélisation. Sciences Sup, dunod édition, Mars 2006. (Cité en pages 42 et 47.)

[CNRS 1988] G.D.R.T.R. CNRS. *Le temps-réel.* Technique et science informatiques, 1988. (Cité en page 3.)

[Cottet 2005] F. Cottet et E. Grolleau. Systèmes temps-réel de contrôle-commande : conception et implémentation. Dunod, 2005. ISBN 2100078933. (Cité en pages 1 et 4.)

[Cucu 2006] L. Cucu et E. Tovar. *A framework for response time analysis of fixed-priority tasks with stochastic inter-arrival times.* ACM SIGBED Review, vol. 3, no. 1, 2006. (Cité en page 32.)

[Cucu 2009] L. Cucu. *Probabilistic real-time schedulability analysis : from uniprocessor to multiprocessor when the execution times are uncertain.* Rapport technique, INRIA, May 2009. (Cité en page 31.)

[Dertouzos 1974] M. Dertouzos. *Control robotics : the procedural control of physical processors.* In IFIP Congress, pages 807–813, 1974. (Cité en pages 9 et 35.)

[Dertouzos 1989] M.L. Dertouzos et A.K. Mok. *Multiprocessor on-line scheduling of hard Real-Time tasks.* IEEE transactions on Software Engineering, vol. 15, no. 12, pages 1497–1506, 1989. (Cité en pages 9 et 47.)

[Diaz 2002] J. Diaz, D. Garcia, K. Kim, C. Lee, L. Bello, L. J.M et O. Mirabella. *Stochastic analysis of periodic real-time systems.* In 23^{rd} IEEE Real-Time Systems Symposium(RTSS02), pages 289–300, 2002. (Cité en page 32.)

[Donnelly 1995] C. Donnelly et R. Stallman. *Bison 1.25 : The YACC-compatible Parser Generator.* Présentation sur internet, November 1995. http ://langevin.univ-tln.fr/CDE/LEXYACC/bison.html. (Cité en page 177.)

[Elloy 1997] J.-P. Elloy. *Systèmes réactifs, Synchronisme, Tâches et évènements.* In École d'été Temps-réel'97, pages 28–37, ECN-IRCyN BP 92101 44321 Nantes Cedex 3, Septembre 1997. Jean-Pierre.Elloy@lan.ec-nantes.fr. (Cité en page 1.)

[El.Rewini 1995] H. El.Rewini et H.H Ali. *Static scheduling of conditional branches in parallel programs.* Parallel and Distributed Computing, vol. 24, pages 41–54, January 1995. (Cité en page 29.)

[Grolleau 1999] E. Grolleau. *Ordonnancement temps-réel optimal à l'aide des réseaux de Petri en environnement monoprocesseur et multiprocesseur.* PhD thesis, LISI, ENSMA, Novembre 1999. (Cité en pages 17, 18, 42, 47, 48, 102, 148, 150, 153, 163 et 187.)

[Grolleau 2002] E. Grolleau et A. Choquet-Geniet. *Off-line computation of real-time schedules using Petri nets.* In Discrete Events Dynamic Systems (DEDS), volume 12, pages 311–333. Kluwer Academic Publishers, Jully 2002. (Cité en pages 9, 17, 42, 47, 48, 51 et 112.)

[Jensen 1997] K. Jensen. *Coloured Petri nets. Basic concepts, analysis methods and Practical Use.* In theoretical Computer Science. Springer Verlag, 1997. (Cité en pages 42 et 43.)

[Kim 1980] K.H. Kim et M. Naghibdadeh. *Prevention of task overruns in real-time non-preemptive multiprogramming systems.* In Perf., pages 267–276, 1980. (Cité en page 6.)

[Kopetz 1995] H. Kopetz. *Why Time-Triggered Architecture will Succeed in Large Hard Real-Time Systems.* In Proceedings of the 5^{th} IEEE Worshop on Future Trends of Distributed Computing Systems, pages 2–9, Chenju, Korea, August 1995. (Cité en page 2.)

[Laalaoui 2005] Y. Laalaoui. *Ordonnancement temps-réel hors-ligne à l'aide des colonies de fourmis en environnement monoprocesseur.* PhD thesis, Institut National d'Informatique, Algerie, 2005. (Cité en page 47.)

[Labetoulle 1974] 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. (Cité en page 35.)

[Largeteau 2002] G. Largeteau et D. Geniet. *Validation Temporelle d'Applications Temps-Réel Distribuées à Contraintes Strictes.* In Real-Time Systems, Paris, France, 2002. teknea, I.S.B.N. : 2-87717-081-0. (Cité en pages 9, 18, 47 et 112.)

[Largeteau 2005] G. Largeteau et D. Geniet. *Discrete geometry applied in hard real-time systems validation.* In Lecture Notes in Computer Science, editeur, 12^{th} Discrete Geometry for Computer Imagery, volume 3429, pages 23–33. Springer-Verlag, 2005. (Cité en pages 18 et 47.)

[Lawler 1981] E.L. Lawler et C.U. Martel. *Scheduling periodically occuring tasks on multiple processors.* Information Processing Letters, vol. 12, pages 9–12, February 1981. (Cité en page 47.)

[Leung 1980] J. Leung et M. Merrill. *A note of preemptive scheduling of periodic real-time tasks.* Information Processing Letters, vol. 11, no. 3, pages 115–118, 1980. (Cité en pages 9, 110 et 111.)

[Leung 1982] J. Leung et J. Whitehead. *On the complexity of fixed priority scheduling of periodic real-time tasks.* Performance Evaluation, pages 237–250, 1982. (Cité en pages 9, 110 et 111.)

[Liu 1973] C.L. Liu et J.W. Layland. *Scheduling algorithms for multiprogramming in real-time environnement.* Journal of the ACM, vol. 20, no. 1, 1973. pp 46-61. (Cité en pages 4, 6, 7, 9, 24, 27, 31, 32, 35 et 39.)

[Lombardie 2010] M. Lombardie, M. Milano, M. Ruggiero et L. Benini. *Stochastic allocation and scheduling for conditional task graphs in multi-processor systems-on-chip.* Journal of Scheduling, vol. 13, no. 4, pages 315–345, 2010. (Cité en pages 27 et 28.)

[Madhukar 2008] A. Madhukar, E. Arvind, F. Sebastian et L. Insup. *Compositional Feasibility Analysis for Conditional Real-Time Task Models.* In Proceedings of the 11^{th} IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC 2008, Orlando, Florida, May 2008. (Cité en page 35.)

[Mar] *Le profil MARTE.* Présentation sur le site web. http ://www.omgmarte.org. (Cité en page 25.)

[Martel 1982] C.U. Martel. *Preemptive scheduling with release times, deadlines and due times.* ACM Computer Journal, vol. 29, pages 812–829, July 1982. (Cité en page 47.)

[Mok 1983] A. K. Mok. *Fundamental design problems of distributed systems for the hard real-time environment.* PhD thesis, Massachussets Institute of Technologie, 1983. (Cité en pages 6 et 32.)

[OMG 2001] OMG. *Unified Modeling Language Specification, V1.4.* Rapport technique, Object Management Group, 2001. (Cité en page 25.)

[Out] *Les outils basés AADL.* Présentation sur le site web. http ://www.aadl.info/aadl/currentsite/tool/. (Cité en page 24.)

[Pailler 2004] S. Pailler et A. Choquet-Geniet. *Off-line scheduling of real-time applications with variable duration tasks.* In WODES : Workshop on Discrete Event Systems, Reims, France, September 2004. Kluwer Academic Publisher. (Cité en page 53.)

[Pailler 2006] S. Pailler. *Analyse hors ligne d'ordonnançabilité d'applications temps-réel comportant des tâches conditionnelles et sporadiques.* PhD thesis, ENSMA : Ecole Nationale Supérieure de Mécanique et d'Aérotechnique, Octobre 2006. (Cité en pages 17, 18, 30, 42, 47, 53, 104, 113, 121, 148, 149, 150, 161, 163, 168, 187 et 188.)

[Peterson 1981] J.L. Peterson. Petri nets theory and the modeling of systems. Prentice-Hall, 1981. (Cité en page 42.)

[Petri 1962] C.A. Petri. *Kommunikation mit automaten.* Rapport technique 1, Bonn Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, New York, 1962. Second Edition, English translation. (Cité en page 42.)

[Phan 2004a] T. H. Phan. *Analyse d'ordonnanabilité des applications temps-réel modélisées en UML.* PhD thesis, Université d' Evry, Juillet 2004. (Cité en page 25.)

[Phan 2004b] T.H. Phan, S. Gerard et F. Terrier. *Real-time system modeling with ACCORD/UML methodology : illustration through an automotive case study.* In Languages for system specification. Kluwer Academic Publishers, 2004. ISBN :1-4020-7990-7. (Cité en page 25.)

[Puaut 2007] I. Puaut. *Méthodes de calcul de WCET : état de l'art.* In ETR 2007, 2007. pp 173-189. (Cité en pages 7 et 13.)

[Puschner 1997] P.P. Puschner et A.V. Schedl. *Computing maximum task execution times. A graph-based approach.* In Real-Time Systems, volume 13, pages 67–91. Kluwer Academic Publishers, 1997. Manufactured in the Netherlands. (Cité en page 7.)

[Pushner 1989] P. Pushner et C. Koza. *Calculating the maximum execution time of real-time programs.* Real-Time Systems, vol. 1, pages 159–176, 1989. (Cité en page 7.)

[Ramchandani 1974] C. Ramchandani. *Analysis of Asynchronous concurrent systems by Petri nets.* Rapport technique, MIT, Cambridge, February 1974. (Cité en page 43.)

[Renault 2009a] X. Renault, F. Kordon et J. Hugues. *Adapting models to model checkers, a case study : Analysing* `AADL` *using Time or Colored Petri Nets.* In 2009 IEEE/IFIP International Symposium on Rapid System Prototyping, pages 26–33, 2009. DOI 10.1109/RSP.2009.30. (Cité en page 24.)

[Renault 2009b] X. Renault, F. Kordon et J. Hugues. *From AADL Architectural Models to Petri Nets : Checking Model Viability.* In 2009 IEEE International

Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, pages 313–320, 2009. DOI 10.1109/ISORC.2009.11. (Cité en page 24.)

[SAE 2008] SAE. *AADL Standard, V2*. Rapport technique, Society of Automotive Engineers, November 2008. (Cité en page 24.)

[Santoshkumar 1998] I. Santoshkumar, G. Manimaran et C. Siva Ram Murthy. *Static scheduling of object-based real-time task with probabilistic conditional branches in distributed systems*. In 6^{th} IEEE International Worshop on Parallel and Distributed Real-Time Systems, Florida, USA, 1998. (Cité en page 29.)

[Serlin 1972] O. Serlin. *Scheduling of time critical processes*. In Spring Joint Computers Conference, pages 925–932, 1972. (Cité en pages 35 et 39.)

[Sgroi 1999] M. Sgroi, L. Lavagno, Y. Watanabe et A. Sangiovanni-Vincentelli. *Quasi-static Scheduling of Embedded Software Using Equal Conflict Nets*. In ICATPN'99, LNCS1639. Springer Verlag, Berlin Heidelberg, 1999. Edited by S. Donatelli and J. Kleijn. (Cité en page 26.)

[Shin 2003] D. Shin et J. Kim. *Power-Aware Scheduling of Conditional Task Graphs in Real-Time Multiprocessor Systems*. In ISLPED'03, Seoul, Korea, August 2003. ACM. (Cité en pages 27 et 28.)

[Sperberg-McQueen] C. M. Sperberg-McQueen et Henry Thompson. *XML Schema*. Présentation sur internet. http ://www.w3.org/XML/Schema. (Cité en page 178.)

[Sprunt 1989] B. Sprunt, L. Sha et P. Lehoczky. *Aperiodic task scheduling for hard real-time systems*. Real-Time Systems Journal, vol. 1, no. 1, pages 27–60, June 1989. (Cité en page 6.)

[Stankovic 1988] J.A. Stankovic. *Misconceptions About Real-Time Computing : A Serious Problem For Next Generation Systems*. IEEE Computer Magazine, vol. 21, no. 10, 1988. pp 10, 19. (Cité en page 1.)

[Stankovic 1998] J.A. Stankovic, M. Spuri, K. Ramamritham et G.C. Buttazo. Deadline scheduling for real-time systems. Kluwer Academic Publishers, 1998. (Cité en page 48.)

[Starke 1990] P. H. Starke. *Some properties of timed nets under the earliest firing rule*. Lecture Notes in Computer Science, vol. 424, no. 5, pages 418–432, 1990. (Cité en page 43.)

[Strosnider 1995] J.K. Strosnider, J.P. Lehoczky et L. Sha. *The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments*. IEEE Transactions on Computers, vol. 44, no. 1, January 1995. (Cité en page 6.)

[Tindell 1992] K.W. Tindell, A. Burns et A.J. Wellings. *Allocating hard real-time tasks : an NP-hard problem made easy*. Real-Time Systems, vol. 4, no. 2, pages 145–165, 1992. (Cité en page 47.)

[Touzalin 2001] I. Queteuil Touzalin. *Analyse comportementale de systèmes complexes critiques par produit synchronisé d'abstractions en réseaux de Petri ordinaires d'agents temps-réel.* PhD thesis, Université de Paris 11, Orsay, France, 2001. (Cité en page 36.)

[Valk 1981] R. Valk et G. Vidal-Naquet. *Petri nets and regular languages.* Journal of Computer and System Sciences, 1981. (Cité en page 44.)

[Verhoosel 1996] J.P.C. Verhoosel, L.R. Welch, D.K. Hammer et E.J. Luit. *Incorporating temporal considerations during assignment and pre-run-time scheduling of objects and processes.* Parallel and Distributed Computing, vol. 36, no. 1, pages 13–31, Jully 1996. (Cité en page 29.)

[Vern 1995] P. Vern. *Flex 2.5 : A fast scanner generator.* Présentation sur internet, March 1995. http ://langevin.univ-tln.fr/CDE/LEXYACC/flex.html. (Cité en page 177.)

[Wilhelm 2008] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat et P. Stenstrom. *The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools.* ACM Trans.Embedd.Comput.Syst., vol. 7, no. 3, april 2008. (Cité en pages 7, 13 et 68.)

[Xu 1992] J. Xu et D.L. Parnas. *Pre-runtime scheduling of processes with exclusion relations on nested or overlapping critical sections.* In Conference on Computers and Communications, Phoenix, USA, April 1992. pp 6471, 6479. (Cité en pages 9, 47 et 112.)

[Xu 1993] J. Xu. *Multiprocessor scheduling of processes with release times, deadlines, precedence and exclusion relations.* IEEE Transactions on Software Engineering, vol. 19, no. 2, pages 139–153, February 1993. (Cité en page 47.)

[Zamorano 1997] J. Zamorano, A. Alonso et J.D.L. Puente. *Exhaustive computation of the scheduled task execution sequences of a real-time application.* In WRTP'97, pages 145, 151, 1997. (Cité en pages 9, 47 et 112.)

Index

Résumé : Nous étudions la modélisation et la validation hors-ligne des applications temps-réel en environnement monoprocesseur. Ces applications sont composées d'un ensemble de tâches. Dans notre étude, nous prenons en compte explicitement l'échange des messages, le partage des ressources et les instructions conditionnelles. Notre objectif est de mettre en évidence, puis de prendre en compte, les relations sémantiques, induites par les tests conditionnels, pouvant exister entre les tâches. Ces relations sont d'une part, les relations à l'intérieur d'une tâche, qui concernent les corrélations entre les comportements choisis dans des conditionnelles successives, portant sur une même valeur, et d'autre part les relations entre plusieurs tâches, qui proviennent du fait que, si elles examinent le même paramètre, afin de déterminer leur comportement, alors ceux-ci seront dépendants.

Classiquement, les applications qui comportent des tests conditionnels sont modélisées de façon linéaire, en encapsulant les blocs conditionnels, et les séquences d'ordonnancement sont utilisées pour leur validation. Cette approche ne permet pas d'obtenir et de valider l'ensemble des comportements effectifs de l'application. Nous proposons dans ce travail une approche de modélisation et de validation arborescente, qui permet de considérer de façon explicite les blocs conditionnels. Nous construisons donc et analysons les arbres d'ordonnancement. Ces arbres permettent de valider les applications sur les plans comportementaux et sémantiques (en représentant tous les comportements effectifs), structurels (en vérifiant que toutes les primitives temps-réel sont prises en compte dans le bon ordre) et temporels.

Nous comparons ensuite ces deux approches de validation, et prouvons que les approches de validation linéaire sont parfois trop pessimistes, c'est à dire qu'elles peuvent conduire à déclarer certaines applications comme non ordonnançables, alors qu'en réalité elles sont ordonnançables lorsqu'on utilise les arbres d'ordonnancement et les approches arborescentes. L'approche arbre est donc parfois nécessaire.

En première approche, nous construisons un générateur qui prend en entrée l'ensemble des codes des instances des tâches de l'application, et fournit en sortie l'ensemble de tous les arbres d'ordonnancement valides sur les plans comportementaux et sémantiques. Ensuite nous en éliminons les instances non valides structurellement ou temporellement. La complexité du générateur étant exponentielle en fonction du nombre de tâches de l'application, cette approche est difficile à mettre en œuvre dans la pratique.

C'est l'une des raisons pour lesquelles nous proposons et validons une approche de modélisation basée sur les réseaux de Petri colorés avec ensemble terminal et fonctionnant sous la règle du tir maximal, qui permet de gérer les contraintes sémantiques, structurelles et temporelles qui nous intéressent. Le réseau construit sera utilisé pour générer les arbres valides, par construction du graphe de marquages, et la complexité pourra être réduite grâce à des heuristiques.

Mots clés : Test conditionnel, Réseau de Petri, Arbre, Sémantique, Validation

Abstract : We study the modeling and the off-line validation of real-time applications in uniprocessor environment. These applications are composed of a set of tasks. In our study, we consider explicitly the exchange of messages, resource sharing and conditional statements.
Our goal is to identify and take the semantic relations into account, induced by conditional tests, which may exist between tasks. These relations are, on the one hand, the relations within a task, concerning relations between behaviors chosen in successive conditional, on a same value, and, on the another hand, the relations between several tasks, which result from the fact that if they examine the same parameter, to determine their behavior, then they will be interdependent.
Classically, applications that contain conditional tests are modeled linearly, encapsulating the conditional blocks, and the scheduling sequences are used for validation. This approach does not allow to obtain and validate all of the effective behaviors of the application. We propose in this work an arborescent approach for modeling and validation, which allow us to consider explicitly the conditional blocks.
So we construct and analyze the scheduling trees. These trees are used to validate the application on the behavioral and semantic plans (by representing all the effective behaviors), the structural plans (by making sure that all real-time primitives are taken in the correct order into account) and the temporal plans.
We then compare these two approaches of validation, and we prove that the linear validations approaches may be too pessimistic, meaning they can lead to declare certain applications such as non schedulable, when in fact they are schedulable when scheduling trees and tree approaches are used. The tree approach is therefore sometimes necessary.
In the first approach, we build a generator that takes as input the set of instances of application tasks, and outputs the set of all valid scheduling trees on the behavioral and semantic plans. Then we eliminate the invalid instances structurally or temporally. The complexity of the generator being exponential in the number of application tasks, this approach is difficult to implement in practice.
This is one reason why we propose and validate an arborescent approach for modeling, based on colored Petri nets with terminal set, and operating under the maximal firing rule, which allow us to manage the semantic, structural and temporal constraints. The Petri net constructed will be used to generate the valid scheduling trees, by construction of the markings graph, and the complexity can be reduced by heuristics.
Keywords : Conditional test, Petri net, Tree, Semantics, Validation

Résumé : Nous étudions la modélisation et la validation hors-ligne des applications temps-réel en environnement monoprocesseur, qui prend explicitement en compte l'échange des messages, le partage des ressources et les instructions conditionnelles entre les tâches. Notre objectif est de mettre en évidence l'impact de ces paramètres sur l'analyse des applications. Classiquement, ces applications sont modélisées de façon linéaire, en encapsulant les blocs conditionnels, et les séquences sont utilisées pour leur validation. Nous proposons une approche de modélisation et de validation arborescente, qui permet de considérer de façon explicite les blocs conditionnels, et qui utilise les arbres d'ordonnancement pour la validation. Nous comparons ensuite ces deux approches, et prouvons que les premières sont parfois trop pessimistes, c'est à dire qu'elles peuvent conduire à déclarer certaines applications comme non ordonnançables, alors qu'en réalité elles le sont. Nous commençons par construire un générateur d'arbres d'ordonnancement valides. La complexité du générateur étant exponentielle en fonction du nombre de tâches, cette approche est difficile à mettre en œuvre dans la pratique. Nous proposons donc une approche de modélisation basée sur les réseaux de Petri. Ce réseau sera utilisé pour générer les arbres valides, par construction du graphe de marquages, et la complexité pourra être réduite grâce à des heuristiques.
Mots clés : Test conditionnel, Réseau de Petri, Arbre, Sémantique, Validation

Abstract : We study the modeling and the off-line validation of real-time applications in uniprocessor environment, wich take explicitly the exchange of messages, resource sharing and conditional statements into account. Our goal is to highlight the impact of these parameters on the analysis of applications. Classically, these applications are modeled linearly, encapsulating the conditional blocks, and the scheduling sequences are used for validation. We propose an arborescent approach for modeling and validation, which allow us to consider explicitly the conditional blocks, and use the scheduling trees for validation. We then compare these two approaches of validation, and we prove that the linear validations approaches may be too pessimistic, meaning they can lead to declare certain applications such as non schedulable, when in fact they are schedulable. We beging by building a generator of valid valid scheduling trees. The complexity of the generator being exponential in the number of application tasks, this approach is difficult to implement in practice. So, we propose an approach for modeling, based on Petri nets. This Petri net will be used to generate the valid scheduling trees, by construction of the markings graph, and the complexity can be reduced by heuristics.
Keywords : Conditional test, Petri net, Tree, Semantics, Validation