Modélisation de la prise en compte de la sémantique dans les applications temps-réel: concepts et outils

Christian Fotsing
LISI, ENSMA
fotsingc@ensma.fr

Annie Geniet
ENSMA, Université de Poitiers
annie.geniet@univ-poitiers.fr

Guy Vidal-Naquet
SUPELEC, Université Paris-Sud
Guy.Vidal-Naquet@supelec.fr

Résumé

Nous posons les bases formelles d'une modélisation des applications temps-réel qui intègre des informations d'ordre sémantique et dont l'objectif est l'analyse d'ordonnancement de ces applications. Nous considérons les tâches comportant des instructions conditionnelles. Nous avons montré que la prise en compte explicite de ces instructions et de la sémantique des tests était importante pour optimiser les possibilités d'ordonnancement. Nous proposons donc une modélisation formelle des systèmes temps-réel basée sur les réseaux de Petri (**RdP**), qui permet en particulier la gestion de la sémantique des tests et des instructions conditionnelles. Le modèle de **RdP** utilisé est un réseau autonome coloré avec marquages terminaux et fonctionnant sous la règle du tir maximal. Nous montrons enfin que le modèle proposé respecte bien les contraintes étudiées et modélise de façon plus réaliste les systèmes temps-réel.

1 Introduction

1.1 Contexte de l'étude

Nous considérons des applications temps-réel de contrôle commande à contraintes strictes. La sécurité du procédé contrôlé dépend fortement de l'adéquation entre la vitesse de réaction du système informatique et celle de l'évolution du procédé. Le non respect des contraintes temporelles de l'application peut avoir des conséquences inacceptables (pertes en vie humaine ou matérielles importantes).
Une application temps-réel est constituée d'un ensemble de tâches. Les tâches manipulées sont des tâches périodiques à départs simultanés ou différés. Elles peuvent se synchroniser, partager des ressources ou échanger des messages. Un enjeu majeur est de s'assurer que les tâches concernées respectent leurs contraintes temporelles. C'est l'objet de la validation temporelle que nous étudions ici. Valider un

système de tâches consiste en particuler à produire un ordonnancement valide du système. De façon plus précise, il s'agit de déterminer une politique d'allocation du (des) processeur (s) aux tâches, i.e. une politique d'ordonnancement pour laquelle on peut garantir que toutes les échéances sont respectées.
Notre objectif est de proposer une méthode formelle orientée modèle de détermination et de simulation d'une stratégie d'ordonnancement en environnement monoprocesseur.

1.2 Approches classiques

Une tâche périodique $T_i = < r_i, C_i, D_i, P_i >$ est classiquement modélisée par quatre paramètres temporels : sa date de réveil r_i qui correspond à sa date de première activation, sa pire durée d'exécution C_i qui est le temps le plus long d'utilisation du processeur par la tâche, son délai critique D_i qui est la durée maximale autorisée entre l'activation et la terminaison de chacune des instances de la tâche et sa période P_i. La tâche est donc réactivée à chaque instant $r_i + k \times P_i$, où k est un entier naturel.
Si le facteur d'utilisation $U = \sum_{i=1}^{n} \frac{C_i}{P_i}$ d'une application est strictement plus grand que 1, elle n'est pas ordonnançable [1]. D'autre part, il suffit de trouver une séquence d'ordonnancement qui respecte ces échéances pour conclure que le système est ordonnançable.
Quand le système temps-réel considéré possède des tâches avec des instructions conditionnelles, les approches de modélisation classiques encapsulent les instructions conditionnelles qui sont représentées par leur plus longue branche qui correspond à la plus longue durée d'exécution de la tâche sur le processeur (on parle du pire cas).
Dans le cas où la tâche contient des primitives temps-réel, les modélisations classiques supposent qu'elles s'exécutent toutes sur cette unique branche de durée maximale, quand bien même ce n'est pas le cas [2] [3] [4]. On modélise donc la tâche par une unique séquence linéaire dont la durée est la pire durée de la tâche.
Les approches classiques amènent donc à un surdimensionnement du matériel et se basent sur un modèle qui ne correspond pas au fonctionnement de l'application. On ne peut donc pas en déduire de simulation fidèle du comportement effectif.
Certains auteurs ont proposé des approches de modélisation qui considèrent de façon explicite les instructions conditionnelles.
Dans [5] les tâches sont représentées sous la forme d'un diagramme **état-transition**. Chaque branche d'une conditionnelle est modélisée comme une succession d'états et de transitions. L'étude de faisabilité du système est ensuite réalisée sur le graphe obtenu en appliquant les techniques de programmation linéaire (**ILP**[1]).
Dans [6], les tâches sont subdivisées en **sous-tâches**. On accède ainsi de façon explicite aux branches conditionnelles. Ensuite la **DBF**[2] est utilisée pour valider le système.

[1] Integer Linear Programming
[2] Demand Bound Function

Remarquons que ces approches 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.
La méthode présentée dans [7], basée sur les réseaux de Petri et modélisant explicitement les instructions conditionnelles, est utilisée pour générer des séquences d'ordonnancement valides. Le système est modélisé comme un ensemble de places et de transitions, et un algorithme d'ordonnancement (*Quasi-static scheduling*) est proposé pour obtenir les séquences valides.
En dehors du modèle de réseau de Petri utilisé, la principale différence, du point de vue méthodologie, avec notre approche est que nous générons, non pas une séquence (linéaire) d'ordonnancement, mais un arbre d'ordonnancement qui exprime mieux le comportement de l'application temps réel, et qui est plus adapté quand les tâches considérées ont des instructions conditionnelles.

1.3 Notre approche

L'un des objectifs de ce papier est de proposer un modèle fin, en vue d'un ordonnancement réaliste, qui prend explicitement en compte les instructions conditionnelles, et fournit une description détaillée du comportement de l'application. Pour cela, nous étendons le modèle de [8] en proposant de considérer un modèle de tâche arborescent (par opposition au modèle linéaire classique) qui met clairement en évidence les instructions conditionnelles et les primitives correspondantes. Ce modèle considère de façon explicite tous les chemins effectivement parcourus par la tâche. Les primitives temps-réel mises en jeu sont donc effectivement représentées sur les branches correspondantes.
De plus, nous prenons en compte la sémantique des tests. Il s'agit ici de tenir compte des relations d'incompatibilité pouvant exister entre les tests conditionnels des différentes tâches du système qu'on veut modéliser.
Notons que dans le cas où aucune tâche ne possède d'instructions conditionnelles, notre modèle est équivalent à celui de [8].
Afin de mieux appréhender les différents modèles que nous manipulons dans ce papier, nous présentons à la Figure 1 un exemple de pseudo code d'une tâche comportant deux instructions conditionnelles, la modélisation classique où on suppose que la ressource **R** est utilisée entre les instants *trois* et *quatre*, et que la tâche a une durée d'exécution égale à *sept*, et la modélisation arborescente où l'on voit que la ressource n'est utilisée que dans une branche, de durée *cinq*, et que la tâche a *trois* comportements de durées respectives *sept*, *six* et *cinq*.
Le modèle proposé est utilisé pour déterminer les ordonnancements valides. Nous redéfinissons la notion d'ordonnancement en remplaçant la notion de séquence (valide) par la notion d'arbre d'ordonnancement (valide) [9]. On a ainsi une notion d'ordonnancement à deux dimensions, dans laquelle on peut effectivement voir et comprendre tous les comportements du système.

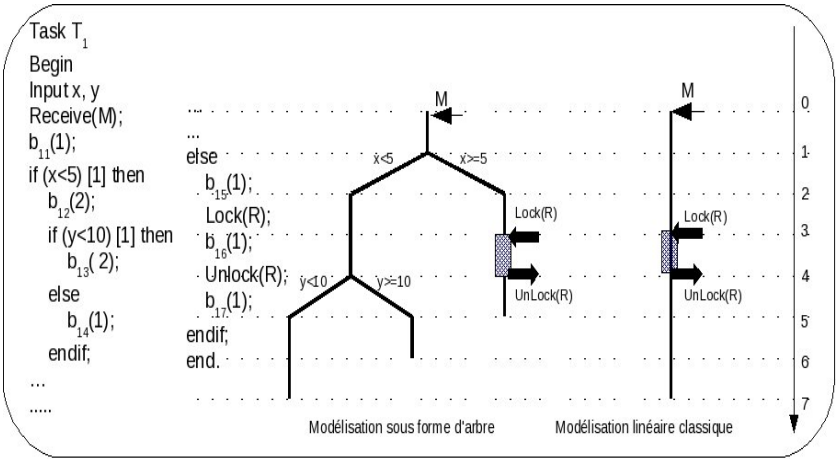

FIG. 1 – Tâche T_1 : Représentation linéaire et arborescente

1.4 Motivations des choix

L'intérêt de notre approche est que l'on considère explicitement tous les comportements des tâches, et donc de l'application. Il est ainsi possible d'en faire une étude détaillée. De plus, nous pouvons tenir compte lors de l'analyse, de la sémantique contenue dans les tests, puisqu'ils sont explicitement décrits. De ce fait, les conclusions d'ordonnançabilité s'appuient sur un modèle plus réaliste, et sont plus fines. Un système peut être déclaré non valide par l'approche classique (encapsulation des conditionnelles) et se révéler valide lorsqu'on affine l'étude à l'aide de notre approche [10].
Pour illustrer nos propos, nous considérons l'exemple extrait de [10] constitué de trois tâches interactives dont les représentations linéaires et arborescentes sont données Figure 2. Il s'agit d'un système simplifié de contrôle commande des essuie-glaces d'un véhicule :

1. la tâche $T_{AcquiredOrder}$ lit l'état du *comodo* et envoie les informations à la tâche $T_{WipesController}$;
2. $T_{WipesController}$ 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_{Order} ;
3. T_{Order} coordonne le balayage du pare brise par les essuie-glaces.

Classiquement, en encapsulant les instructions conditionnelles et en considérant les plus longues branches pour chaque tâche, le système n'est pas ordonnançable, car le facteur d'utilisation est plus grand que 1 ($U = \sum_{i=1}^{3} \frac{C_i}{P_i} = \frac{12}{11} > 1$).
Une étude plus détaillée, qui considère les comportements effectifs de l'application, montre que le cas correspondant à $U > 1$ est un cas impossible, parce qu'on ne peut pas avoir simultanément $contact = off$ et $contact = on$. L'arbre d'or-

donnancement que nous donnons à la Figure 3 est un arbre valide, et le système initial est en fait ordonnançable.

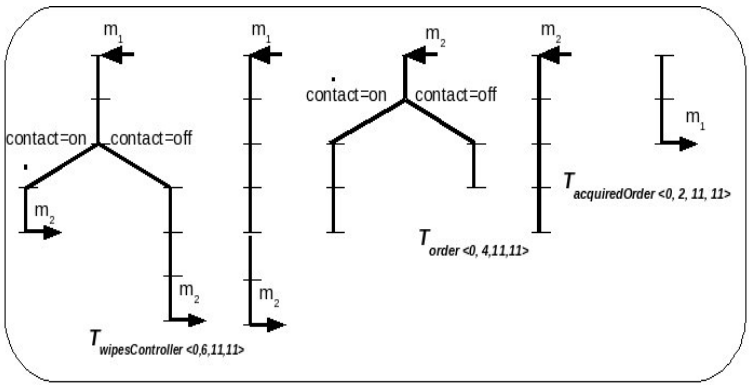

FIG. 2 – Représentation linéaire et arborescente des tâches d'un système simplifié de contrôle commande des essuie-glaces d'un véhicule

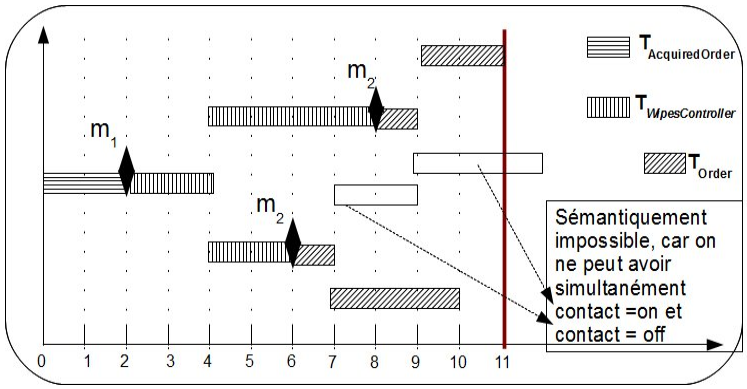

FIG. 3 – Un arbre d'ordonnancement valide du système de la Figure 2

1.4.1 Du choix des approches hors ligne

Les approches hors ligne ont une puissance d'ordonnancement supérieure aux approches en ligne. En effet, dans le cas géneral, il n'existe pas de stratégies en ligne optimale [11] (i.e. qui ordonnance correctement toute application ordonnançable), car le problème de l'ordonnancement est NP-dur [12]. Par contre, les approches hors ligne 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, le problème de la complexité de l'énumération des séquences valides peut se gérer grâce à de bonnes heuristiques pendant la recherche

des séquences valides.
Nous choisissons dans ce papier d'utiliser les approches d'ordonnancement hors ligne. Notre approche s'appuie sur une modélisation basée sur les **RdP** qui, comme nous l'avons dit, étend la modélisation proposée dans [8].

1.4.2 Du choix de l'outil de modélisation

Le choix de cet outil de modélisation provient de plusieurs motivations : l'implémentation assez flexible, la possibilité de modéliser aisément l'échange de messages, le partage de ressources, le parallélisme, la concaténation et le choix d'instructions, la représentaion facile des blocs pré-emptifs ou non, l'obtention d'un graphe de marquage qui correspond à l'ensemble des comportements valides quand un bon alphabet est défini sur le réseau, à partir duquel on peut générer de façon automatique un ordonnancement valide du système de tâches modélisé.
Ce document présente un modèle formel qui répond à nos besoins : modélisation détaillée des tâches, de leurs contraintes temporelles, des instructions conditionnelles et de la sémantique des tests. Ce modèle servira de base à une analyse complète d'ordonnancement. Nous justifions ensuite la modélisation adoptée.
Le papier se présente de la façon suivante : nous posons tout d'abord les hypothèses adoptées et les définitions nécessaires pour comprendre notre étude, ensuite nous proposons le modèle formalisant notre approche, puis enfin nous le justifions. Nous terminons en illustrant notre approche sur un exemple. Dans la conclusion, nous montrons comment nous comptons utiliser le modèle pour la génération des arbres d'ordonnancement.

2 Hypothèses liées à l'étude et définitions

Nous supposons qu'une application temps-réel est définie comme un ensemble $\mathcal{S} = \{T_i, i = 1...n\}$ de tâches périodiques. Ces tâches peuvent comporter des instructions conditionnelles et utiliser des primitives temps-réel.
Dans toute la suite, $|X|$ désigne le cardinal de l'ensemble X.
Nous faisons l'hypothèse que formellement, une tâche est constituée :

1. de blocs séquentiels ne contenant ni primitives temps-réel, ni conditionnelles. $Block_i = \{b_{i1}(l_1) \dots b_{i|Block_i|}(l_{|Block_i|})\}$ est l'ensemble des blocs d'exécution de la tâche T_i. $b_{ik}(l_k)$ designe le k^{eme} bloc de la tâche T_i de durée d'exécution l_k. Ces durées peuvent être obtenues par une approche de calcul statique ou dynamique [13]. Notre étude ne concerne pas le calcul de cette pire durée d'exécution. Nous la supposons donc acquise.
2. de conditionnelles *if* $Test_{ij}$ *then . . . else . . . endif.*
   $Test_i = \{Test_{i1}, \overline{Test_{i1}} \dots Test_{i|Test_i|}, \overline{Test_{i|Test_i|}}\}$ est l'ensemble des tests qui interviennent dans la tâche T_i. $\overline{Test_{ij}}$ désigne la négation de $Test_{ij}$ et $\overline{\overline{Test_{ij}}} = Test_{ij}$. L'ensemble des tests de l'application est $Test = \bigcup_{i=1}^{n} Test_i$. Nous notons $Test_{ij}^k$ la k^{eme} occurence du j^{eme} test qui se produit au sein de

l'instance $T_{i,k}$ de la tâche T_i. Nous associons une durée d'exécution δ_{ik} à chacun des tests $Test_{ik}$ et $\overline{Test_{ik}}$;

3. de primitives temps-réel :
   (a) de gestion de ressources. $\mathcal{R} = \{R_1 \dots R_{|R|}\}$ est l'ensemble des ressources utilisées par l'application.
   $\mathcal{INS}_R = \{Lock(R_r), Unlock(R_r)/R_r \in \mathcal{R}\}$ est l'ensemble de toutes les instructions qui portent sur les ressources. Les règles suivantes assurent la cohérence de la gestion des ressources :
      i. si une ressource est prise dans une branche d'une conditionnelle, elle est libérée dans cette même branche ;
      ii. si elle est prise à l'extérieur d'une branche d'une conditionnelle, elle ne peut pas être libérée au sein de cette branche.

   De plus l'instruction $Lock(R)$ signifie qu'un verrou est placé sur la ressource R et $Unlock(R)$ viendra le lever ;
   (b) de gestion de messages. $\mathcal{M} = \{M_1 \dots M_{|M|}\}$ est l'ensemble des messages échangés dans l'application.
   $\mathcal{INS}_M = \{Send(M_m), Receive(M_m)/M_m \in \mathcal{M}\}$ est l'ensemble de toutes les instructions qui portent sur les messages.
   $Send(M)$ signifie qu'une tâche dépose un message M dans une boite à lettre B implicite ici, et ce message sera récupéré grâce à l'instruction $Receive(M)$. L'envoi des messages n'est pas bloquant, mais la réception l'est.
   Afin d'éviter la perte des messages, qui peut entraîner des anomalies d'ordonnancement, nous ne traitons que des systèmes dans lesquels les messages émis sont effectivement reçus. Remarquons que cette condition n'impose pas que deux tâches qui échangent des messages aient la même période, mais que les fréquences d'émission et de réception doivent être les mêmes.
   De plus, les émissions et les réceptions de messages ne doivent pas se faire à l'intérieur des blocs conditionnels. En effet, s'il y'a émission dans un bloc conditionnel, la tâche réceptrice risque de rater son échéance. De même s'il y'a réception dans un bloc conditionnel, certains messages risquent de n'être jamais reçus.

   $\mathcal{INS}_R \cup \mathcal{INS}_M$ constitue l'ensemble des primitives temps-réel. Nous supposons que les durées d'exécution des primitives temps-réel sont contenues dans les durées des blocs d'exécution adjacents.

Les hypothèses considérées dans cette étude 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.
Une tâche T_i est complètement modélisée par :

1. son arbre d'exécution qui représente tous les comportements possibles de la tâche. Formellement, c'est un arbre dont les liens sont étiquetés par des lettres de l'alphabet $\otimes_i = \mathcal{INS}_R \cup \mathcal{INS}_M \cup Block_i \cup Test_i$, et dont les liens partant des noeuds doubles sont étiquetés par deux éléments $Test_{ij}$ et $\overline{Test_{ij}}$ de $Test_i$;
2. r_i sa date de première activation ;
3. ζ_i un multi-ensemble de durées, chaque durée correspondant à la durée de l'un des comportements de la tâche ;
4. D_i son délai critique ;
5. P_i sa période.

Une tâche T_i consiste en une infinité d'instances $T_{i,k}$, $k \in \mathbb{N}^*$. $T_{i,k}$ est activée à l'instant $r_{i,k} = r_i + (k-1) * P_i$ et a comme échéance absolue $d_{i,k} = r_i + (k-1) * P_i + D_i$.
La notion de chemin permet d'isoler un comportement de la tâche.

Définition 1 *Un chemin P_{il} d'une tâche T_i est une branche de son arbre d'exécution. On note $Path_i$ l'ensemble des chemins de la tâche T_i. Un chemin P_{il} est un mot de taille finie écrit sur l'alphabet $\otimes_i = \mathcal{INS}_R \cup \mathcal{INS}_M \cup Block_i \cup Test_i$. On note C_{il} la durée d'exécution du chemin P_{il}.*

Afin de repérer les incohérences sémantiques à l'intérieur d'un chemin, nous définissons une relation d'incompatibilité entre deux tests de $Test_i$.
Nous considérons au niveau de la relation d'incompatibilité les tests de type seuillage ($Temperature \leq 5$) ou identification ($x = 3$). Ces tests portent sur des paramètres qui sont fournis aux tâches en entrée, et ne sont donc pas modifiés durant toute une période par la tâche. Remarquons que d'autres types de tests peuvent exister, mais leur sémantique n'est pas étudiée ici.

Définition 2 *La relation d'incompatibilité $\mathcal{RIT}_i$ est définie sur $Test_i$ par :*
si $t, t' \in Test_i$, $t\ \mathcal{RIT}_i\ t'$ si et seulement si $t^k \Rightarrow \overline{t'}^k$, $k \in \mathbb{N}^*$.

Par exemple, si $t_1 = (x < 5)$ et $t_2 = (x > 10)$ alors $t_1 \Rightarrow \overline{t_2}$.
Nous nous sommes restreints à des tests de seuillage et d'égalité. Nous pouvons donc faire une analyse syntaxique du code des tâches, afin de trouver l'ensemble des tests incompatibles.
Nous définissons ensuite les notions de chemins effectifs à l'intérieur d'une tâche, puis de chemins incompatibles entre deux tâches.

Définition 3 *Un chemin P_{il} d'une tâche T_i est un chemin effectif si et seulement si il ne contient pas deux éléments t et t' de $Test_i$ tels que $t\ \mathcal{RIT}_i\ t'$. On note $Pathef_i$ l'ensemble des chemins effectifs de T_i.*

Dans la suite, le terme chemin désignera un chemin effectif.
L'ensemble de tous les chemins effectifs de l'application temps-réel est l'ensemble $Pathef = \bigcup_{i=1}^{n} Pathef_i$.

Définition 4 *Nous étendons la relation d'incompatibilité $\mathcal{RIT}_i$ sur l'ensemble $Test$ en définissant la relation d'incompatibilité $\mathcal{RIT}$ par :*
$\forall t, t' \in Test$, t $\mathcal{RIT}$ t' si et seulement si :

1. *si $t \in Test_i$ et $t' \in Test_j$ alors $r_i = r_j$ et $P_i = P_j$;*
2. *$\forall k \in \mathbb{N}^*$, $t^k \Rightarrow \overline{t'}^k$.*

Le premier point permet de rendre cohérentes les mises en relation des tests, instance par instance. En effet, pour comparer deux tests appartenant à deux tâches différentes, il faut s'assurer que la valeur de la variable de tests n'a pas été modifiée, afin qu'on ne puisse que comparer des variables comparables. La comparaison est possible ici parce qu'on a supposé que les tests portaient sur des paramètres en entrée (**IN**) non modifiables.
Enfin, nous définissons une relation d'incompatibilité entre les chemins des différentes tâches.

Définition 5 *Deux chemins $P_{il} \in Pathef_i$ et $P_{i'l'} \in Pathef_{i'}$ ($i \neq i'$) sont incompatibles si et seulement si $\exists t \in Test_i$ et $t' \in Test'_i$ tel que t apparaît dans P_{il}, t' apparaît dans $P_{i'l'}$ et t $\mathcal{RIT}$ t'. On le note P_{il} $\mathcal{RI}$ $P_{i'l'}$.*

Notons qu'il est possible de construire l'ensemble $\mathcal{RI}$ par analyse syntaxique du code de l'application temps-réel.
Nous présentons dans la Section 3 le modèle retenu pour modéliser et ordonnancer les applications temps-réel.

3 Modélisation par RdP

Le **RdP** [14] que nous proposons pour la modélisation de $\mathcal{S}$ est défini par le couple (N, M_0) où N = (Q, T, W) avec Q l'ensemble des places, T l'ensemble des transitions, M_0 le marquage initial du réseau et W : QxT $\cup$ TxQ $\Rightarrow \mathbb{N}^*$ la fonction de valuation. Il s'agit d'un réseau coloré (nous utilisons un réseau coloré simplifié avec deux couleurs **a** et **b**) fonctionnant sous la règle du tir maximal : on tire des ensembles maximaux de transitions valides, avec un ensemble terminal qui définit les propriétés que doit vérifier tout marquage accessible.
La règle du tir maximal permet de modéliser le temps, le marquage initial permet en particulier de prendre en compte les dates de première activation, et l'ensemble terminal permet de gérer les contraintes liées aux échéances.
Notre **RdP** comporte trois parties (Figure 11) : le système temporel pour la modélisation de la dynamique du système, la couche sémantique pour la modélisation des relations d'incompatibilité entre les tests des instructions conditionnelles et de la cohérence sémantique du système et le système de tâches pour la représentation du système de tâches.

3.1 Le système de tâches

Une tâche est constituée d'un ensemble de chemins qui interagissent. Pour modéliser un chemin, nous utilisons la modélisation proposée dans [8] (Figure 5). Elle consiste en une succession finie de places et de transitions à laquelle nous ajoutons l'ensemble des tests du chemin. Les tests sont modélisés comme un ensemble de blocs indépendants dont la durée est égale à la durée du test. Chaque chemin correspond à un chemin de l'arbre d'exécution, donc il est associé à une valeur pour chacun des tests rencontrés. Par exemple, pour le système de la Figure 10, le chemin P_{11} correspond aux valeurs "then" des deux tests rencontrés.
Soit $i \in 1 \ldots n$, **une tâche** T_i est modélisée par l'ensemble de ses chemins. Ils partagent tous la place $Activ_i$ qui les met en exclusion mutuelle. **L'ensemble des places** est $Q_{P_i} = \{Activ_i, P_{ijkl}/k \in 1 \ldots 3, j \leq C_{il}\ et\ l \in 1 \ldots |Path_i|\}$:

1. $Activ_i$ gère l'activation de la tâche T_i et de la non réentrance ;
2. P_{ijkl} designe la j^{eme} place de la tâche T_i se trouvant sur le chemin l. *k* est utilisé pour la compression des places, afin d'éviter une explosion en nombre de places de notre modélisation. La modélisation d'un bloc de durée d respecte le schéma de la Figure 4.

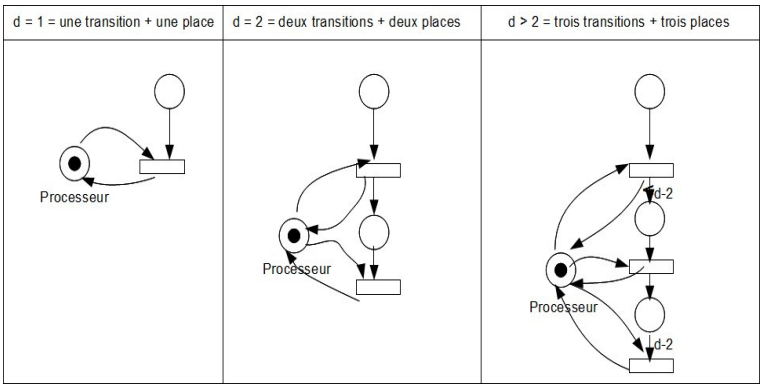

FIG. 4 – modélisation d'un bloc de durée d

L'ensemble des transitions est
T_{P_i} = $\{T_{ijkl}/k \in 1 \ldots 3, j \leq C_{il}$ et $l \in 1 \ldots |Path_i|\}$: T_{ijkl} designe la j^{eme} transition de la tâche T_i se trouvant sur le chemin l. *k* est défini selon le schéma de la Figure 4.
Il faut ensuite gérer l'échange de messages et le partage des ressources entre les tâches d'un même système.
L'ensemble Q_I = $\{Processor, R_r, M_m/r \in 1 \ldots |R|\ et\ m \in 1 \ldots |M|\}$ est celui des places nécessaires pour ces échanges :

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 ;

3. chaque place M_m correspond à un message, donc à une boite aux lettres. Il y a autant de places M_m que de messages échangés.

La fonction de valuation W est définie sur cette partie par :

1. W($Activ_i$, T_{i11l}) = **a** + **b**, où **a+b** est l'ensemble $\{a, b\}$;
2. W(Fin, $Activ_i$) = **b**, pour chacune des dernières transitions Fin des différents chemins (pour marquer la fin de l'exécution de l'instance en cours) ;
3. W(T_{ijkl}, $Processor$) = 1 et W($Processor$, T_{ijkl}) = 1. On modélise 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, si une ressource R_r est prise au début du bloc j, puis est libérée à la fin du bloc j' du chemin P_{il} d'une tâche T_i ;
5. pour une tâche T_i qui envoie un message M_m sur le chemin $P_{i,l}$ à une tâche $T_{i'}$ sur le chemin $P_{i',l'}$, on a W(T_{ijkl}, M_m) = 1 si un 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. $M_0(Activ_i)$ = **a+b** si r_i = 0 et $M_0(Activ_i)$ = **b** si $r_i > 0$. Nous supposons ici que $r_i \leq P_i$;
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 $|R_r|$ est le nombre d'instances de la ressource R_r ;
5. $M_0(M_m)$ = 0. Les boites aux lettres sont vides au départ.

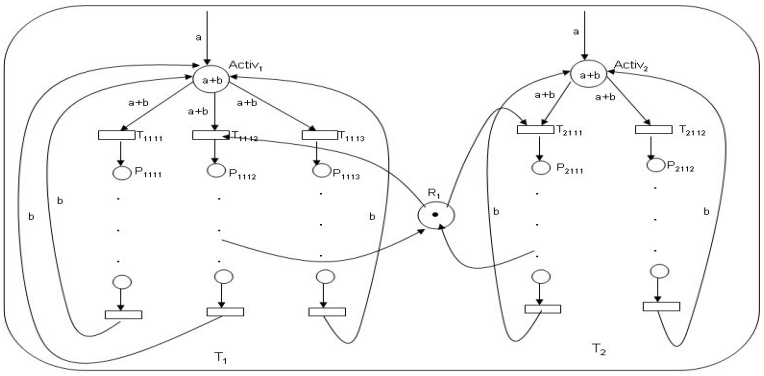

FIG. 5 – Modélisation d'un système de deux tâches : la première a *trois* chemins et la deuxième en a *deux*. De plus, elles partagent une ressource R_1. Pour des raisons de lisibilité, la place $Processor$ n'est pas représentée ici

3.2 Le système temporel

Il permet de gérer le temps. Nous ajoutons des places et des transitions qui permettent de modéliser l'écoulement du temps (Figure 6).
L'ensemble des transitions ajoutées est $T_T = \{RTC, Clk_i / i \in 1 \dots n\}$:

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 les places $Time(i)$. 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 dans la place $Activ_i$.

L'ensemble des places mises en oeuvre dans la gestion de la périodicité des tâches est $Q_T = \{Time(i) / i \in 1 \dots n\}$. 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. On complète la **fonction de valuation** :

1. W(Time(i),Clk_{il}) = P_i (pour que les tâches soient activées périodiquement) ;
2. W(RTC, $Time(i)$) = 1.

De même, on ajoute au **marquage initial** :
$M_0(Time(i)) = P_i - r_i + 1$ si $0 < r_i \leq P_i$ et $M_0(Time(i)) = 1$ si $r_i = 0$. Ce qui permet de gérer les dates de réveils, ceux ci pouvant être simultanés ou différés.
Afin de modéliser les delais critiques, nous définissons l'ensemble terminal $\mathcal{I}$ par :

$$M \in \mathcal{I} \Leftrightarrow \begin{cases} 1.\ M(Time(i)) > D_i \Rightarrow M(Activ_i) = \mathbf{b}; \\ 2.\ M(Time(i)) = 1 \Rightarrow M(Activ_i) = \mathbf{a} + \mathbf{b}\ ou\ M(Activ_i) = \mathbf{b}; \\ 3.\ M(Time(i)) \leq P_i. \end{cases}$$

Les deux premiers points formalisent les contraintes qui assurent le respect des délais critiques, et le troisième point garantit la réactivation périodique des tâches.

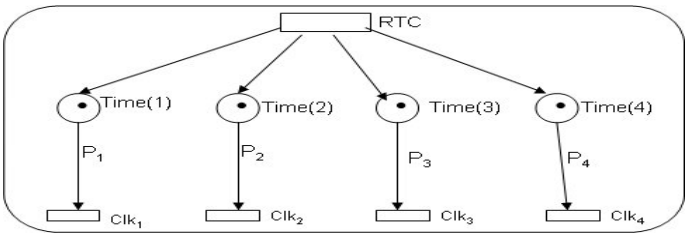

FIG. 6 – Modélisation de la couche temporelle : $(Time(i), Clk_i)$ modélise l'horloge locale de la tâche T_i, et RTC représente l'horloge (globale) du système.

3.3 La couche sémantique

L'aspect sémantique concerne la prise en compte des relations d'incompatibilité entre chemins. Il s'agit de garantir l'exclusion mutuelle entre deux chemins

incompatibles, tout en garantissant que tous les comportements cohérents seront préservés.
Pour chaque couple $(P_{il}, P_{i'l'}) \in \mathcal{RI}$ ($1 \leq i < i' \leq n$, $l \in 1 \ldots |Path_i|$ et $l' \in 1 \ldots |Path_{i'}|$) de chemins incompatibles, on crée une place $Excl_{ili'l'}$ qui contient initialement un jeton, et qui est reliée aux premières transitions de ces chemins. Le tir de ce jeton détermine le chemin choisi, et un seul chemin peut être choisi à la fois.
Le système de gestion des exclusions mutuelles doit être initialisé à chaque période. Pour cela, on associe à chaque chemin P_{il} une place Sem_{il} initialement vide. Le tir de T_{i11l} y dépose un jeton (Figure 7).

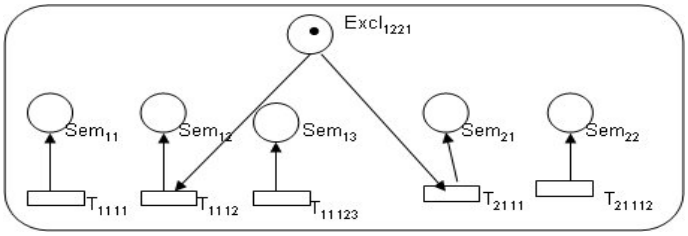

FIG. 7 – Modélisation de l'exclusion mutuelle : les chemins P_{12} et P_{21} sont incompatibles

On remplace ensuite la transition initiale Clk_i qui était associée à chaque tâche T_i par un ensemble de transitions $\{Clk_{il}/l \in 1 \ldots |Path_i|\}$. Le tir de Clk_{il} néccessite P_i jetons dans $Time(i)$ et un jeton dans Sem_{il}, et il dépose un jeton dans toutes les places d'exclusion en entrée de T_{i11l} ($Excl_{il..}$ ou $Excl_{..il}$) (Figure 8).

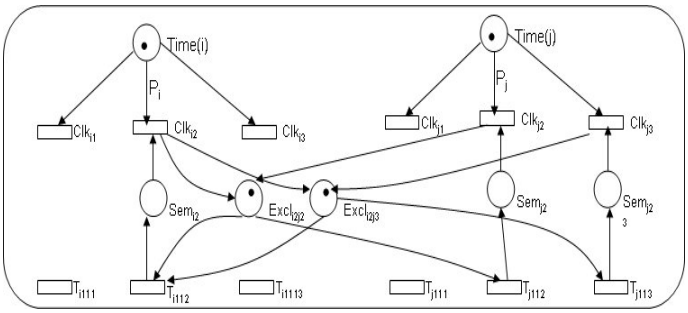

FIG. 8 – Gestion de la cohérence sémantique : les couples de chemins (P_{i2}, P_{j2}) et (P_{i2}, P_{j3}) sont incompatibles

Dans le cas où les dates de réveil sont nulles, l'activation des tâches se fera sans soucis, parce que les places $Activ_i$ ont initialement **a+b** jetons. Si les dates de réveil des tâches sont non nulles, la place $Activ_i$ contient initialement **b** jetons et les places Sem_{il} 0 jeton.
Pour qu'une première activation soit possible, on complète donc la couche sémantique en ajoutant à chaque tâche T_i une place Sem_{i0} marquée à 1 et une transition

Clk_{i0} (Figure 9).

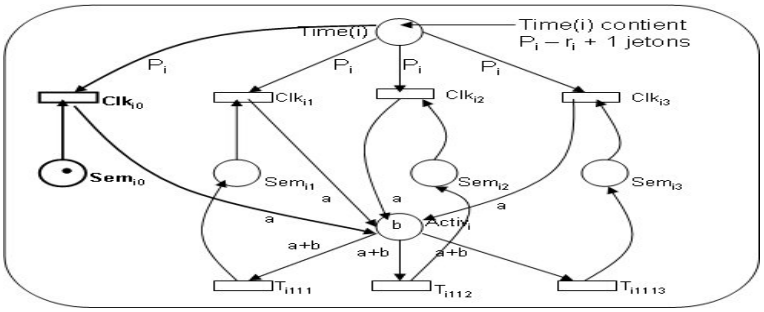

FIG. 9 – Complétion de la couche sémantique pour une tâche T_i quand $r_i > 0$

On complète 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}, $Activ_i$) = **a** (l'activation d'une tâche dépose un jeton **a** dans la place $Activ_i$) et W($Time(i)$, Clk_{i0}) = P_i.

De même, on ajoute au **marquage initial** : $M_0(Sem_{il}) = 0$ pour $l > 0$, $M_0(Sem_{i0}) = 1$ et $M_0(Excl_{ili'l'}) = 1$ pour le couple de chemins incompatibles P_{il} et $P_{i'l'}$.

4 Validation du modèle

Nous souhaitons valider notre modèle. pour cela, nous voulons montrer que :

1. on ne peut pas exécuter dans une même période deux chemins incompatibles ;
2. aucun comportement n'est definitivement perdu : à chaque activation, chaque tâche dispose bien de tous ses chemins effectifs.

Deux invariants sont vérifiés par notre système.

Invariant 1 $\forall i, i' \in 1 \ldots n, \forall l \in 1 \ldots |Path_i|, \forall l' \in 1 \ldots |Path_{i'}|, M(Sem_{il}) + M(Excl_{ili'l'}) + M(Sem_{i'l'}) = 1$

L'invariant 1 assure que deux chemins incompatibles ne peuvent pas être choisis pendant la même période.

Démonstration 1 *On fait une preuve par induction. 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 est vrai pour un marquage accessible M quelconque. Soit σ *un ensemble maximal de transitions tiré, qui fait passer le système dans un état M'. Montrons 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, on avait $M(Excl_{ili'l'}) > 0$ (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 $Processeur$ 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 $T_{i'11l'} \in \sigma$, on utilise le même raisonnement qu'au point $2.(a)$;*
   (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 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 $Clk_{i'l'} \in \sigma$, on utilise le même raisonnement qu'au point $2.(c)$.*

Invariant 2 $\forall i \in 1 \ldots n, \forall l \in 1 \ldots |Path_i|, \sum_{l=1}^{|Path_i|} M(Sem_{il}) \leq 1$ *et*
$\forall i \in 1 \ldots n$, ***M(**$Activ_i$**) = a+b*** $\Rightarrow \forall l \in 1 \ldots |Path_i|$, ***M(**$Sem_{il}$**) = 0***

L'invariant 2 garantit qu'à l'issue de la réactivation, tous les chemins peuvent être choisis. Le système ne perd donc pas de possibilités.

Démonstration 2 *Comme précédemment, on fait une preuve par induction.*
Par définition $\sum_{l=1}^{|Path_i|} M_0(Sem_{il}) = 0$.
On suppose maintenant que les propriétés vérifiées 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, $Trans = \{T_{i11l}, Clk_{il}\}$:

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, toutes les places $Sem_{i,l}$ 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 $\sum_{l=1}^{|Path_i|} M'(Sem_{il}) = 1$;*

*(b) M($Activ_i$) ≠ **a+b**, donc, 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, $\sum_{l=1}^{|Path_i|} M'(Sem_{il}) = 0$.*

On ajoute à ces invariants la propriété suivante, qui garantit la cohérence sémantique de l'application.

Propriété 1 $\forall i \in 1 \dots n, \forall l \in 1 \dots |Path_i|, \forall l' \in 1 \dots |Path_{i'}|, M(Time(i)) = 1 \Rightarrow M(Excl_{ili'l'}) = 1$

Démonstration 3 *Soit $Excl_{ili'l'}$ une place d'exclusion mutuelle.*
Si $M(Time(i)) = 1$, par définition de la relation d'incompatibilité, on a $r_i = r_{i'}$ et $P_i = P_{i'}$, donc on a $M(Time(i')) = 1$.
De même, si on vient de réactiver la tâche T_i (en tirant la transition Clk_{ik}), on a réactivé la tâche $T_{i'}$ (en tirant la 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 1). Ainsi, à l'issue du tir de Clk_{il}, on a consommé la marque de Sem_{il} et déposé une 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 2) :*
   (a) *si $k' = l'$, $Clk_{i'l'}$ transfère une 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 1), et les tirs de Clk_{ik} et $Clk_{i'k'}$ n'ont pas modifiés le marquage de $Excl_{ili'l'}$.*

Remarquons que nos objectifs sont atteints :

1. deux chemins incompatibles P_{il} et $P_{i'l'}$ (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 1 ;
2. à l'issue de la réactivation, tous les chemins d'une tâche peuvent être choisis, car $M(Excl_{ili'l'}) = 1, \forall i, i', l, l'$ (Propriété 1).

En conclusion, l'exclusion mutuelle entre chemins incompatibles est bien respectée, et aucun comportement n'est perdu.

5 Illustration

Le but de cette section est de montrer, sur un exemple illustratif, comment nous utilisons notre modèle pour gérer les tâches conditionnelles, et les éventuelles

relations d'incompatibilités qui peuvent exister dans un système.
Nous considérons le système S constitué de trois tâches conditionnelles, dont les tests portent sur les mêmes variables **Input** x, y et z. Nous présentons à la Figure 10 une illustration des tâches de ce système sous leur forme arborescente et linéaire. L'ensemble des chemins incompatibles $\mathcal{RI} = \{(P_{13}, P_{31}), (P_{14}, P_{21}), (P_{23}, P_{33})\}$.

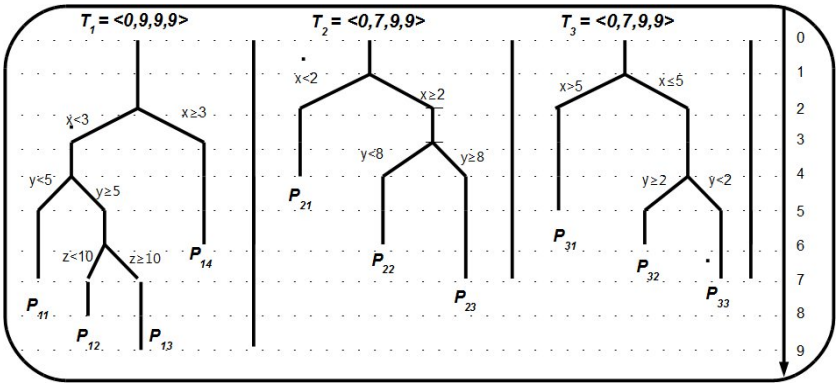

FIG. 10 – Système S : représentation arborescente et linéaire

Le modèle de **RdP** que nous présentons à la Figure 11 représente cette application. Les 10 arcs valués par b, qui relient les dernières transitions aux places $Activ_i$ n'ont pas été représentés. De même, la place $Processor$ et les 74 arcs qui la relient aux autres transitions ne sont pas représentés. On voit sur le modèle les 4 (resp. 3 et 3) chemins de la tâche T_1 (resp. T_2 et T_3), ainsi que les 3 places d'exclusions mutuelles qui modélisent la relation $\mathcal{RI}$. Les tâches sont à départs simultanés, donc les places Sem_{i0} ne sont pas représentées ici.

6 Conclusion

Nous avons proposé une modélisation formelle des applications temps-réel permettant de représenter de manière explicite les instructions conditionnelles. Ce modèle permet de simuler de manière précise les différents comportements effectifs de l'application.
Pour cela, nous avons également modélisé les corrélations existantes entre les tests utilisés dans les différentes tâches : les choix des branches sont alors corrélés.
Nous avons intégré à notre modèle une couche (la couche sémantique) qui permet de ne générer lors des simulations que les comportements de l'application où les comportements individuels des différentes tâches sont compatibles entre eux.
À partir du modèle, nous pouvons construire le graphe des marquages terminaux. Ce graphe est ensuite valué par l'alphabet défini par la fonction

$$\psi : \begin{cases} T_{ijkl} \Rightarrow T_i \\ RTC \Rightarrow \epsilon \\ Clk_{il} \Rightarrow \epsilon \end{cases}$$

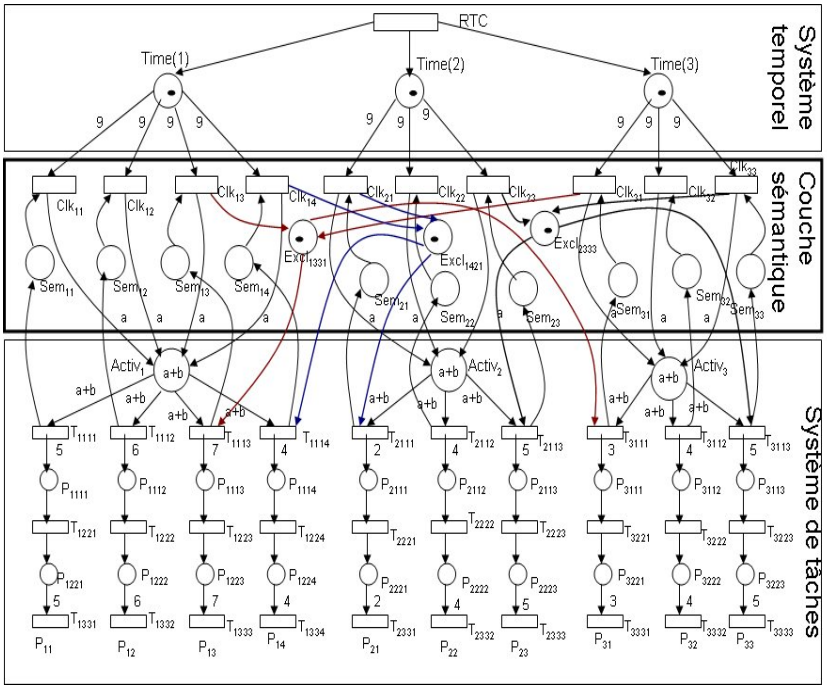

FIG. 11 – Modélisation du système S

où ϵ est un mot vide.
Les arbres d'ordonnancement sont contenus dans ce graphe. La prochaine étape de notre travail consistera en leur extraction.
Par ailleurs, nous pourrons utiliser des critères qualitatifs afin de diriger l'extraction. Nous pourrons par exemple extraire les arbres tels que le temps de réponse moyen de certaines tâches soit minimal (ce qui est intéressant dans une optique de gestion de fautes par mécanisme de reprise), ou bien tels que les temps creux du processeur soient répartis équitablement (ce qui peut permettre une gestion efficace de tâches apériodiques à contraintes strictes).
Notons pour terminer que cette première version du modèle sera optimisée. En effet, nous avons représenté de manière disjointe les différents comportements des tâches. Cette première étape a permis la définition de la couche sémantique, et la justification de la structure. Dans une deuxième version, nous allons modéliser les tâches par des structures arborescentes, plus compactes, donc moins gourmande en place.
Tout ceci sera intégré dans un outil complet, comme décrit à la Figure 12.

Références

[1] Buttazo.G.C. *Hard real Time computing systems : predictable scheduling algorithms and applications*. Kluwer Academic, 1997.

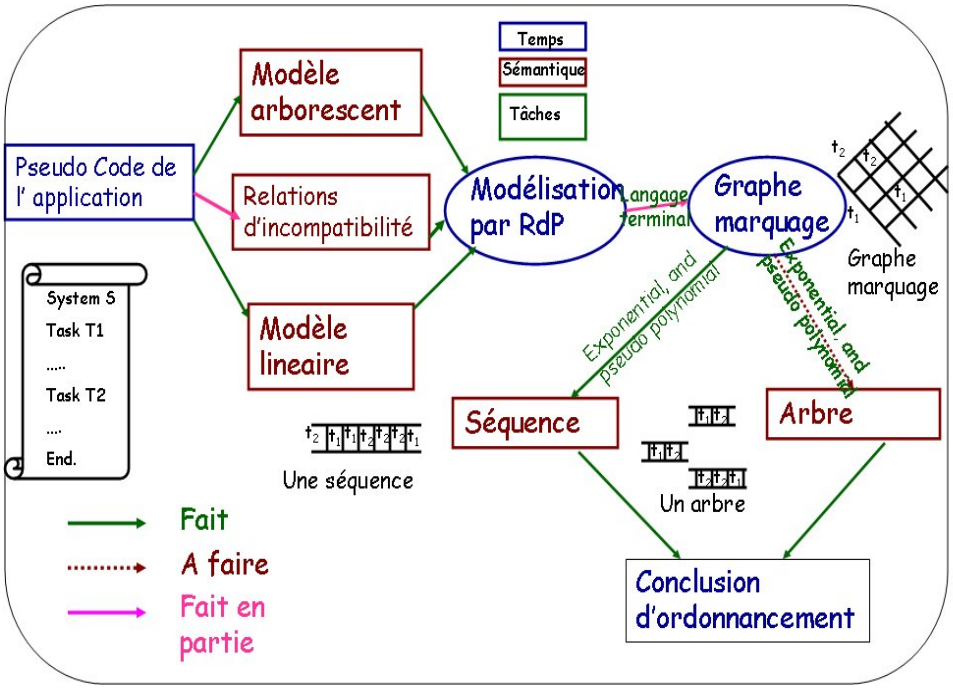

FIG. 12 – Structure de l'outil complet pour l'étude de l'ordonnancement des systèmes temps-réel

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

[3] J.P. Babau. *Étude du comportement temporel des applications temps réel à contraintes strictes basée sur une analyse d'ordonnançabilité*. PhD thesis, University of Poitiers, 1996.

[4] E. Niehaus. Program representation and translation for predictable real-time systems. In 12^{th} *IEEE Real-Time Systems Symposium*, pages 53,63., San Antonio, Texas, 1991.

[5] C. Aussaguès and V. David. A method and a technique to model and ensure timeliness in safety critical real-time systems. In *ICECCS'98*, 1998. 4th International Conference on Engineering of Complex Computer Systems, Monterey, California, U.S.A.

[6] S. Baruah. Dynamic and static priority scheduling of recurring real-time tasks. In *Real-Time Systems*, pages 93–128. Kluwer Academic Publishers, 2003. Manufactured in the Netherlands.

[7] M. Sgroi, L. Lavagno, Y. Watanabe, and 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.

[8] E. Grolleau and A. Choquet-Geniet. Off-line computation of real-time schedules using Petri nets. In *Discrete Events Dynamic Systems*, pages 311–333. Kluwer Academic Publishers, 2002. Manufactured in the Netherlands.

[9] C. Fotsing, A. Geniet, and G. Vidal-Naquet. A realistic modeling of real-time systems for efficient scheduling. In Proceedings of SEW-33, 33rd annual IEEE Software Engineering Workshop in Skôvde, Sweden, October 2009.

[10] C. Fotsing, A. Geniet, and G. Vidal-Naquet. Tree scheduling versus sequential scheduling. In Worshop Volume of ACM Digital Library, ISBN 978-1-60558-915-2, CARS@EDCC, Valencia, Spain, April 2010.

[11] M.L. Dertouzos and A.K. Mok. Multiprocessor on-line scheduling of hard Real-Time tasks. In *Proceedings of IEEE transactions on Software Engineering*, pages 1497, 1506, 1989. 15 (12).

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

[13] 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, and P. Stenstrom. The worst-case execution time problem — overview of methods and survey of tools. *ACM Trans.Embedd.Comput.Syst.*, 7(3), april 2008.

[14] A. Choquet-Geniet. *Les réseaux de Petri, un outil de modélisation*. Sciences Sup, Mars 2006. ISBN 2100491474, Dunod.