Réseaux de Petri temporels à inhibitions/permissions - Application à la modélisation et vérification de systèmes de tâches temps réel

Florent Peres

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *l'Institut National des Sciences Appliquées de Toulouse*
Discipline : Informatique - Sûreté du logiciel et calcul haute-performance

---

Présentée et soutenue par

Florent PERES

le mardi 26 Janvier 2010

Titre :

Réseaux de Petri temporels à inhibitions/permissions

–

Application à la modélisation et vérification de systèmes de tâches temps réel

---

JURY

Président
M. Charles ANDRÉ — Professeur à l'université de Nice Sophia Antipolis

Rapporteurs
M^{me} Béatrice BÉRARD — Professeur à l'université Pierre et Marie Curie
M^{me} Françoise SIMONOT-LION — Professeur à l'école des mines de Nancy

Examinateur
M. Patrick FARAIL — Ingénieur Airbus

Directeurs de thèse
M. Bernard BERTHOMIEU — Chargé de Recherche, LAAS-CNRS
M. François VERNADAT — Professeur à l'Institut National des Sciences Appliquées de Toulouse

---

Ecole Doctorale : Mathématiques, Informatique et Télécommunications de Toulouse
Unité de Recherche : Laboratoire d'Architecture et d'Analyse des Systèmes (LAAS-CNRS) — UPR 8001

A mon père

Si la mort est le but, pourquoi donc sur les routes
Est-il dans les buissons de si charmantes fleurs ?
Et lorsqu'au vent d'automne elles s'envolent toutes,
Pourquoi les voir partir d'un œil momifié de pleurs ?

Si la vie est le but, pourquoi donc sur les routes
Tant de pierres dans l'herbe et d'épines aux fleurs,
Que, pendant le voyage, hélas ! nous devons toutes
Tacher de notre sang et mouiller de nos pleurs ?

Louise-Angélique Bertin

Remerciements ...

La tradition est tenace mais se doit d'être respectée, c'est donc ici que je vais présenter ceux que je veux remercier.

Nombreux remerciements d'abord à François Vernadat, sans qui cette aventure n'aurait peut-être pas eu lieu ou en tout cas aurait certainement été bien différente. Dès lors que j'ai manifesté mon intérêt pour cette thèse, il a tout mis en œuvre pour m'ôter le plus de soucis monétaires possibles, entre le moment où mon DEA se terminait et où ma thèse a officiellement démarré, période qui a tout de même duré plus d'un an. Malgré sa charge de travail, il a toujours su trouver un moment pour discuter. Son esprit toujours ouvert, malgré certaines de mes divagations, il a su canaliser ma soif de nouveauté afin qu'elle ne se disperse pas dans des voies trop diffuses et éparses et au final stériles. Me supporter n'a pas dû toujours être facile, et pour son calme indéfectible, je l'en remercie.

Remerciements également à Bernard Berthomieu, sans qui bien des aspects techniques me sembleraient toujours aussi hermétiques. Il m'a fait découvrir un nouveau langage, m'a permis de me plonger au sein d'un code très complexe, a toujours été disposé à m'écouter même sur des sujets très pointus et grâce à ses connaissances à m'aiguiller efficacement. Je n'ai malgré tout pas suivi sa passion pour Solaris, mais cela m'a permis de confirmer que, décidément oui, l'informatique est plurielle et ne s'arrête pas à qui vous savez et qui vous savez.

Remerciements à l'équipe d'Airbus : Patrick Farail, mon tuteur industriel au cours de cette thèse, Pierre Gaufillet, Jean-Patrice Giacometti, Pascal Thore, Marie-Line Valentin. Nous avons pu échanger des manières de fonctionner différentes ainsi que des discussions souvent passionnées sur des sujets très divers. Ils m'ont permis d'appréhender la complexité liée à la conception de la partie logicielle d'un avion et les problématiques, souvent autres que purement techniques, liées à leurs améliorations.

Remerciements à Charles André, Béatrice Bérard, Françoise Simonot-Lion pour avoir accepté de rapporter ma thèse, pour avoir assisté à ma soutenance et pour m'avoir indiqué des erreurs qui s'étaient glissé dans le manuscrit.

Remerciements à tous les doctorants, post-doctorants, permanents (oui, oui même eux ! :-)) du Laas, qui ont partagé un moment de détente, de travail, un repas, un pot, une soirée, leur maison, que sais-je ? ... avec moi. Merci donc à Karim Guennoun, Mohamad El Masri, Rodrigo Saad, Najla Chamseddine, Ali Kalakech, Alin Stephaniu, Ismael Bouassida, Riad Ben-Halima, Jorge Gomez, German Sancho, Frédéric Nivor, Ihsane Tou, Roxana Albu, Sandy Rahme, Layale Saab, Baptiste Jacquemin, Lionel Bertaux, François Armando, Ion Alberdi, Joan Mazel, Guillaume Dugue, Nouha Abid, Akram Hakiri, Ahmed Akl, Osama Hamouda ... et tous ceux que j'oublie ! Pour ne pas mélanger les torchons et les serviettes (blague, humour), je remercie également Pierre-Emmanuel Hladik, Silvano Dal Zilio, Didier Le Botlan, Jean Fanchon, Serge Bachmann et Gina Briand. Enfin salutations militantes à Laurent Blain, Dimitri Peaucelle et au noyau (qui se reconnaîtra).

Remerciements à mes amis : Jérémie Vergnaud, Mathieu Subra, Emmanuel Briand, Jean-Roch Belin, Mathieu Millet ... où s'arrêter ? comme de bien entendu, ceux que je n'ai pas cité vont m'en vouloir ... merci donc également à Marie, Val, Louise, Mickael, Nathalie, Cynthia, Guillaume, Lucas, Jérémie, Jérémie, Geneviève, Laura, Laura, Guilhem, Brice, Fred, Sacha, Julien, Seb, Cédric, Antoine, Adrien, Damien, Anne, Eric, Déborah, Arnaud, Marie, Magdalena ... pour tous les joyeux moments passés ensembles. Un petit coucou spécial à Yof et papa-Sandra :-)

Remerciements à ma famille qui m'a toujours soutenu dans mes entreprises et dans les moments difficiles. Remerciements spéciaux à ma mère Anne-Marie, à mes sœurs Estelle et Myriam, à ma grand-mère Paulette et à ma tante Jeanine.

... et réciproquement

Puisqu'il faut bien un début à tout, j'inaugure ici une section consacrée aux non-remerciements.

Je ne remercie pas le gouvernement pour sa politique de précarité dans la recherche, elle mérite mieux que ça. Cet entêtement à tout vouloir régenter à l'aune de la concurrence forcenée ne correspond pas à ma vision de la recherche et m'a fait vivement réagir, au détriment de l'avancée du présent travail de thèse. Résultat : plus de trois mois de perdus.

Mention spéciale au cancer, qui en un laps de temps très court a enlevé trop de personnes de mon entourage, dont une très proche, mon père, Maurice. Sa disparition, si soudaine, a crée un vide immense dans ma vie, que rien ne pourra jamais combler. Ce n'est qu'une fois disparus que l'on se rend compte à quel point les gens que l'on aime tiennent une place importante dans nos vies. C'est à sa mémoire que je dédie ce travail, lui qui n'aura pu voir son achèvement.

Tu me manques.

Table des matières

1 Introduction **15**
1.1 Organisation de la thèse 16

2 Eléments contextuels à la vérification par *model checking* **17**
2.1 Etapes du *model checking* 18
2.2 Systèmes de transitions temporisés 18
2.3 Logique Temporelle à temps Linéaire 20
2.4 *Model checking* de *LTL* à l'aide des automates de Büchi 21
2.5 Décidabilité de l'accessibilité des états 22
2.6 Bisimulation 24
2.7 Réseaux de Petri temporels 25
2.7.1 Définitions 27
2.7.2 Sémantique 28
2.7.3 Décidabilité 31
2.8 Des classes pour les *TPN* 33
2.8.1 Graphe de classe linéaire 33
2.9 Extensions 35
2.9.1 read arcs 35
2.9.2 arcs inhibiteurs 35
2.9.3 Arcs chronomètres 36
2.10 Conclusion 37

3 Réseaux de Petri à permissions et inhibitions **39**
3.1 Introduction - Motivation de l'extension 40
3.2 Réseaux de Petri temporels à permissions et inhibitions 42
3.3 Expressivité 43
3.3.1 Introduction 43
3.3.2 Composition des *ipTPN* 45
3.3.3 Les *ipTPN* pour rendre les *TPN* composables 50
3.3.4 Les *ipTPN* pour coder les TA 54
3.4 Abstractions de l'espace d'états pour les *ipTPN* 57
3.4.1 Graphe de classes (à temps) linéaire 57
3.4.2 Graphe de classes linéaires fortes 59
3.5 Conclusions 61

4 Le langage Pola **63**
4.1 Introduction 63
4.1.1 Objectif 63
4.1.2 Un langage spécifique à un domaine 64
4.1.3 Systèmes temps réel 64
4.1.4 Caractérisation du domaine 66
4.1.5 des langages dédiés existants 67

4.2 Description du langage . . . 72
4.2.1 Groupes de tâches et points de réordonnancement à la OSEK . . . 75
4.2.2 partitionnement à la ARINC 653 . . . 83
4.3 Conclusion . . . 89

5 Sémantique de POLA **91**
5.1 Le méta-modèle POLA . . . 91
5.2 Préprocesseur de réseaux de Petri . . . 93
5.3 Notation graphique . . . 94
5.3.1 Associée aux arcs . . . 94
5.3.2 Associée à tout élément . . . 95
5.4 Sémantique par traduction . . . 96
5.4.1 Syntaxe textuelle des réseaux de Petri à permission/inhibition . . . 96
5.4.2 *system* et ressources . . . 96
5.4.3 Traduction d'une tâche . . . 97
5.4.4 Traduction d'une action . . . 97
5.4.5 Traduction d'une période . . . 98
5.4.6 Traduction d'un *offset* . . . 99
5.4.7 Traduction d'une échéance . . . 99
5.5 Traduction d'une politique d'ordonnancement . . . 99
5.5.1 Sur les politiques dynamiques . . . 101
5.6 Traduction d'une allocation . . . 102
5.6.1 Utiliser ou ne pas utiliser la politique d'ordonnancement, telle est la question . . . 103
5.6.2 La traduction . . . 104
5.7 Assemblage des éléments . . . 107
5.7.1 Séquencement et priorité des évènements . . . 107
5.7.2 Gestion de l'indéterminisme des actions / périodes / offsets . . . 109
5.7.3 Composition des éléments POLA . . . 111
5.8 Conclusion . . . 111

6 L'outil POLA : Etudes de cas et expérimentations **113**
6.1 Architecture . . . 113
6.1.1 Propriétés . . . 115
6.2 Présentation des modèles . . . 116
6.2.1 Modèles utilisant des groupes OSEK . . . 116
6.2.2 Base . . . 117
6.2.3 Variation 1 . . . 117
6.2.4 Variation 2 . . . 118
6.2.5 Variation 3 . . . 119
6.2.6 Variation 4, $4'$ et $4''$ . . . 119
6.3 Influences des paramètres . . . 119
6.4 Cas d'étude : ARINC 653 . . . 120
6.4.1 Base . . . 120
6.4.2 Variation 1 . . . 121
6.4.3 Variation 2 . . . 121
6.4.4 Les variations de la variation 3 . . . 121
6.4.5 Résultats . . . 122
6.5 Corrections d'erreurs et raffinages quantitatifs . . . 123
6.5.1 Une erreur dans le modèle OSEK . . . 124
6.5.2 Analyse quantitative manuelle par raffinages successifs . . . 125
6.5.3 Comparaison avec l'outil TIMES . . . 125
6.6 Conclusion . . . 127

7 Conclusion **129**
7.1 Contributions 129
7.2 Perspectives 131

Chapitre 1

Introduction

Un système est constitué d'éléments qui définissent son évolution par leurs interactions. Un système peut être clos : ses éléments n'interagissent qu'avec d'autres éléments du système, ou ouvert : certains éléments interagissent avec l'environnement extérieur au système. Traditionnellement, un programme est un système initialement ouvert, mais clos par ses paramètres (les données utilisateur) et fournissant une réponse, sans effectuer d'interaction avec son environnement (puisqu'il est alors clos), en un temps que l'on souhaite le plus rapide possible. Pour interagir correctement avec l'environnement, un système ouvert doit agir de manière harmonieuse avec celui-ci. Les systèmes temps réel sont des systèmes ouverts contraints par le temps. Lorsqu'un stimulus émane de l'environnement, le système doit le traiter et fournir une réponse dont l'échéance est contrainte dans le temps. Traduit dans le domaine de l'informatique, c'est un programme réactif qui ne doit pas être rapide, mais qui doit répondre au «bon moment» : il interagit avec un environnement qu'il lui est impossible de contrôler, c'est donc à lui de s'adapter.

La vérification formelle des systèmes temps réel demeure un défi lancé à la science. Bien que de nombreux systèmes aient été traités avec succès, la complexité des systèmes conçus actuellement dans les domaines avioniques et automobiles, pour ne citer qu'eux, est encore bien souvent au-delà de la capacité des outils et méthodes déjà existants. L'utilisation des méthodes de vérification formelles telles que le *model checking* est souvent perçue comme difficile et peu adaptée aux usages répandus et admis par la communauté des concepteurs de systèmes temps réel. En effet, les méthodes formelles sont habituellement basées sur un langage pauvre en éléments mais riche en expressivité. L'utilisation de ces langages implique une très bonne connaissance de ces briques primitives, et parfois une remise en question des méthodes de conception connues. Au contraire, les concepteurs de système temps réel utilisent à la fois des concepts de haut et de bas niveau d'abstraction, répandus et éprouvés, qui sont souvent très éloignés sémantiquement des concepts utilisés par les langages formels. Il existe donc un fossé entre les méthodes de conception des systèmes temps réel et les techniques de vérification formelles. Ce fossé est en passe d'être comblé par une tendance à l'utilisation de langages de plus haut niveau, plus proches des attentes des utilisateurs. Mais cette mise au niveau d'exigence des utilisateurs ne va pas sans mal. Les limites théoriques et pratiques des techniques de vérification formelles sont souvent dépassées par les demandes hétérogènes et parfois conflictuelles en terme de niveau d'abstraction.

Cette thèse Cifre – effectuée en partenariat entre Airbus et le Laas-Cnrs – s'intéresse aux éléments caractérisants les systèmes temps réel et à leur description à l'aide d'un langage formel, les réseaux de Petri temporels, couramment admis comme adaptés à cette tâche. Malgré cette reconnaissance commune, la modélisation de systèmes même simples de quelques tâches périodiques ordonnancés de façon basique n'en est pas moins compliquée et peut se révéler être un défi dans certains cas. Nous nous sommes appliqués à l'étude de ce qui rend les systèmes temps réel si difficiles à modéliser.

En premier lieu, nous avons constaté certaines limites des réseaux de Petri temporels en ce qui concerne une modélisation efficace d'une gamme étendue de politiques d'ordonnancements. Ramené à un concept minimal, ceci est dû à leur incapacité à imposer un ordre d'apparition à des évènements concurrents survenant au même instant. De plus, les systèmes temps réel ont une nette tendance à

être constitués d'éléments récurrents composés pour former un système final. Or les réseaux de Petri temporels sont peu adaptés à une utilisation compositionnelle un tant soit peu générale. Ceci nous a donc poussé à proposer une extension des réseaux de Petri temporels permettant de résoudre ces problèmes.

Afin de cerner un périmètre clair d'adéquation de cette nouvelle extension des réseaux de Petri temporels à la modélisation des systèmes temps réel, nous nous sommes concentrés sur la définition d'un langage dédié poursuivant deux objectifs : identifier un sous-ensemble des systèmes temps réel modélisables par cette nouvelle extension et apporter un langage simple à la communauté temps réel permettant par construction et idéalement de manière automatique de vérifier les systèmes écrits à l'aide de ce langage.

1.1 Organisation de la thèse

Pour introduire tous les mots-clés en une phrase, nous dirons que notre thèse va s'intéresser à la *spécification* et à la *vérification* des *systèmes de tâches temps réel* et cela grâce à l'utilisation d'un langage de spécification *dédié*, appelé Pola et à une technique de vérification automatique, le *model checking* par le biais d'une extension des réseaux de Petri temporels, les *réseaux de Petri temporels à inhibitions/permissions*.

Cette thèse se situe donc au confluent des problématiques de vérification et de modélisation. Nous présenterons dans le premier chapitre ce contexte, selon ces deux perspectives. Le point de vue de la modélisation, en introduisant le langage formel des réseaux de Petri temporels, dont nous étudierons l'adéquation à la modélisation des systèmes temps réels. Nous donnerons également le contexte de vérification dans lequel nous nous plaçons, qui est la technique de vérification automatique des modèles, dite *model checking*. Cette technique implique des bonnes propriétés sur la représentation du comportement, permettant l'utilisation de classes de logiques. Le chapitre 2 sera consacré à la présentation de ce contexte.

Le travail effectué dans cette thèse se résume en deux points principaux :

Une extension des réseaux de Petri temporels. Les réseaux de Petri temporels permettent déjà une grande expressivité en regard de la spécification des systèmes, souvent réactifs, contraints par le temps. Malgré cela, certaines caractéristiques temps réel résistent à une modélisation préservant la bisimulation temporisée faible, c'est-à-dire que les possibilités en termes de branchements ne sont pas tout le temps modélisables. Nous verrons que deux nouvelles relations entre les transitions étendent la gamme des systèmes modélisables en tenant compte de la préservation des propriétés de branchements lors de la spécification des caractéristiques problématiques ci-dessus mentionnées. Nous verrons également que cette nouvelle extension permet une concision accrue, point non négligeable dans la guerre face à l'explosion combinatoire, mais surtout redonne la liberté de composition parallèle naturelle des réseaux de Petri, perdue avec l'extension temporelle. Cette extension est étudiée dans le chapitre 3.

Un langage vérifiable *dédié à la spécification des systèmes de tâches temps réel, nommé* Pola. Ce langage a plusieurs objectifs : il donne la puissance de vérification liée à l'utilisation des réseaux de Petri tout en prenant pour hypothèse que les utilisateurs du langage ne doivent pas connaître les mécanismes de cette vérification. Cela n'est possible que si le processus de vérification est automatisable : le *model checking* remplit le mieux cette condition. Ensuite, il donne un cadre permettant de démontrer/étudier les forces et les limites des réseaux de Petri (étendus de diverses manières, dont celle proposée dans cette thèse) face à la modélisation de systèmes de tâches temps réel complexes.

Le 4ème chapitre est consacré à la présentation de ce langage, où sa syntaxe sera donnée au fur et à mesure de la modélisation de deux cas d'études issus des besoins industriels. Puis sa sémantique sera présentée dans le chapitre 5 et des résultats d'expérimentation de la chaîne de traduction le seront dans le chapitre 6.

Chapitre 2

Eléments contextuels à la vérification par *model checking*

Les techniques d'analyses disponibles permettant d'assurer la correction du fonctionnement d'un système sont au moins au nombre de trois. La preuve de la correction peut être complète ou partielle selon la technique utilisée.

Les techniques de simulation, ou de test [Ber07] Tester (resp. simuler) un système signifie exécuter le système dans un environnement réel (resp. abstrait). L'exécution est contrôlée par l'injection de signaux simulant l'environnement. La réaction du système est ensuite examinée par le contrôleur, déterminant ainsi si le comportement est correct. Les points forts de l'analyse par tests sont l'automatisation de la technique, la possibilité de vérifier le code au niveau le plus concret : la machine et le faible coût en ressource nécessaire. Un point négatif important vient contrebalancer ces points positifs : il est impossible de couvrir tous les comportements permis, ou de s'assurer que la couverture effectuée est complète. Un système testé correct ne veut pas dire que le système est correct pour toutes les situations possibles, mais seulement pour les situations testées.

Les techniques basées sur la preuve de théorème Un modèle, s'il est écrit dans une notation formelle, est un objet mathématique. Il est donc possible de prouver des théorèmes valides pour celui-ci. Cette technique admet une grande souplesse quant aux propriétés vérifiables et permet également de couvrir tous les comportements d'un seul coup ou seulement une partie de ceux-ci en fonction de la propriété vérifiée. Cette technique n'est pas automatisable de manière générale à cause de la limitation théorique de la décidabilité des propriétés, mais il existe des assistants de preuve performants capables de générer automatiquement une grande partie de la preuve et de laisser les parties trop complexes à l'intelligence humaine. Ces preuves restent malgré tout très difficiles d'accès pour une personne non-initiée et certaines d'entre elles restent complexes pour les personnes les plus expérimentées. L'article [Wie06] fournit un bon aperçu des outils de preuves disponibles.

Les techniques exhaustives de vérification de modèle, ou *model checking* Celles-ci permettent de vérifier des modèles ayant un comportement fini par une couverture complète. Les propriétés qui sont vérifiées sont de type qualitative : elles doivent admettre soit la réponse **oui** soit la réponse **non**. Autrement dit, soit la propriété est valide pour le système, soit elle ne l'est pas. Les propriétés sont vérifiées exhaustivement : si une propriété est vérifiée, elle l'est pour l'ensemble des exécutions possibles du système constituant son comportement. La vérification est entièrement automatisable dans la limite des systèmes décidables, elle ne l'est que partiellement pour les systèmes indécidables, pour lesquels il est impossible d'exhiber une procédure de vérification générale automatique. Outre l'indécidabilité, son autre inconvénient est de nécessiter beaucoup de ressources car le comportement du système — l'espace des états accessibles — doit être stocké en mémoire. Lorsque cet espace d'état est trop gros pour tenir en mémoire, on dit qu'on est confronté à une *explosion*

combinatoire du nombre d'états.

De part son automatisation capable de cacher les détails de son fonctionnement et sa capacité de couverture complète des comportements, c'est la technique du *model checking* qui est le choix le plus évident pour constituer le moteur de vérification d'une chaîne de vérification des systèmes temps réel. Sa limitation aux modèles à comportements finis est bien sûr une limitation importante, mais il est en général possible de se ramener à un comportement fini, soit en utilisant une abstraction, soit en limitant le modèle par des bornes sur les quantités autrement non bornées (ou indécidablement bornées).

2.1 Etapes du *model checking*

L'utilisation de la technique du *model checking* requiert un modèle et des propriétés.

Le comportement du modèle est tout d'abord extrait par un explorateur de comportement. Ce comportement est typiquement représenté sous la forme d'un graphe orienté étiqueté par des informations sur les nœuds et les arcs.

Une propriété est décrite par une formule, elle-même écrite à l'aide d'une logique modale, donnant la façon de parcourir le graphe du comportement afin d'en extraire les informations nécessaires à la validation de la propriété. Selon le type de logique utilisée, la formule peut donner lieu à l'utilisation d'un algorithme différent : pour la logique *CTL* [CE81], qui est une logique permettant de raisonner sur les possibilités de branchements, l'algorithme d'exploration filtre l'ensemble des états accessibles : la formule est dite valide si l'état initial fait partie de ceux filtrés qui la respectent. Ou bien, si l'on utilise la logique *LTL* [Pri67], qui est une logique permettant de raisonner sur les séquences d'exécutions, une représentation sous forme de graphe, homogène avec la représentation du comportement, est calculée pour ensuite être composée avec le comportement du modèle. Cette composition fournie les informations nécessaires à la décision de la validité de la formule.

Lorsqu'une formule est fausse, il faut souvent faire appel à l'expertise de l'utilisateur pour repérer le problème. Afin de lui faciliter la tâche, il est possible de lui fournir une séquence contre-exemple. Lorsque la propriété attendue n'est pas satisfaite par le modèle, il est possible que celle-ci soit trop forte, ou bien que le modèle soit à revoir, voire même une combinaison des deux possibilités. Dans tous les cas, le contre-exemple est une aide non négligeable pour repérer l'erreur et modifier le modèle et/ou la formule. Les étapes du *model checking* sont données par la figure 2.1.

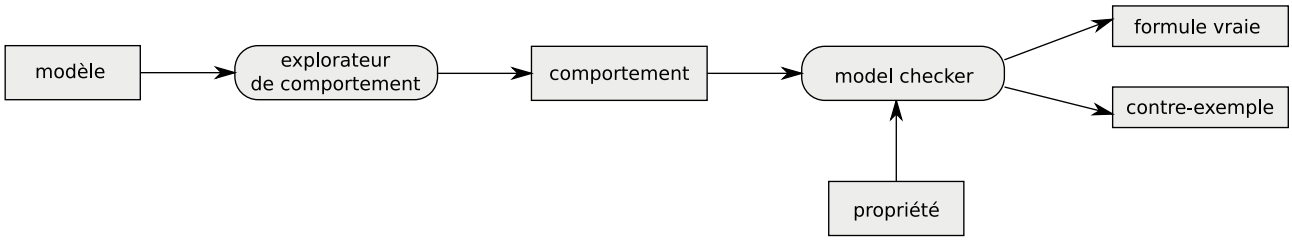


Fig. 2.1 – Etapes de la vérification par *model checking*

2.2 Systèmes de transitions temporisés

Les systèmes de transitions temporisés [HMP92] (abrégé *TTS*) sont une structure offrant la possibilité d'étiqueter les transitions d'une relation de transitions par un scalaire (réel) afin de représenter le passage du temps. Cette structure en forme de graphe orienté (voir figure 2.2) est la base de la plupart des langages formels : tous vont être traduits dans une structure plus ou moins similaire à celle-ci. Nous sommes donc en présence d'une structure représentant l'un des plus bas niveau de la

spécification d'un modèle (pour faire l'analogie avec un ordinateur, les *TTS* peuvent être considérés comme un code machine).

Définition 1 Système de transitions temporisé *Un* système de transitions temporisé *est une structure* $\langle Q, q_0, \Sigma \cup \{\epsilon\}, \rightarrow \rangle$ *où :*

- Q *est un ensemble d'*états *;*
- $q_0 \in Q$ *est* l'état initial *;*
- Σ *est un ensemble fini d'*actions *qui ne contiennent pas l'action* silencieuse ϵ *;*
- $\rightarrow \subseteq Q \times (\Sigma \cup \{\epsilon\} \cup \mathbb{R}^+) \times Q$ *est la relation de* transition.

$(q, a, q') \in \rightarrow$ est noté $q \xrightarrow{a} q'$. Σ^ϵ désigne $\Sigma \cup \{\epsilon\}$. Les transitions dans $Q \times \Sigma^\epsilon \times Q$ sont appelées transitions *discrètes*, celles dans $Q \times \mathbb{R}^+ \times Q$ sont des transitions *continues*. Les transitions continues obéissent aux conditions suivantes $(\forall d, d' \in \mathbb{R}^+)$:

délais nul : $q \xrightarrow{0} q' \Leftrightarrow q = q'$;
additivité : $q \xrightarrow{d} q' \wedge q' \xrightarrow{d'} q'' \Rightarrow q \xrightarrow{d+d'} q''$;
continuité : $q \xrightarrow{d+d'} q' \Rightarrow (\exists q'')(q \xrightarrow{d} q'' \wedge q'' \xrightarrow{d'} q')$;
déterminisme temporel : $q \xrightarrow{d} q' \wedge q \xrightarrow{d} q'' \Rightarrow q' = q''$

Définition 2 Produit de systèmes de transitions temporisés

Soient $S_1 = \langle Q^1, q_0^1, \Sigma_1^\epsilon, \rightarrow_1 \rangle$ *et* $S_2 = \langle Q^2, q_0^2, \Sigma_2^\epsilon, \rightarrow_2 \rangle$ *deux systèmes de transitions temporisés. Le produit de* S_1 *et de* S_2 *est le* TTS $S_1||S_2 = \langle Q^1 \times Q_2,\ q_0^1||q_0^2,\ \Sigma_1^\epsilon \cup \Sigma_2^\epsilon,\ \rightarrow \rangle$, *où* $\rightarrow$ *est la plus petite relation telle que :*

$$\frac{q_1 \xrightarrow{a} q_1' \quad a \in \Sigma_1^\epsilon \setminus \Sigma_2}{q_1||q_2 \xrightarrow{a} q_1'||q_2} \qquad \frac{q_2 \xrightarrow{a} q_2' \quad a \in \Sigma_2^\epsilon \setminus \Sigma_1}{q_1||q_2 \xrightarrow{a} q_1||q_2'} \qquad \frac{q_1 \xrightarrow{a} q_1' \quad q_2 \xrightarrow{a} q_2' \quad a \neq \epsilon}{q_1||q_2 \xrightarrow{a} q_1'||q_2'}$$

La figure 2.2 donne un système de transitions temporisé. Cette figure ne donne qu'une représentation finie d'une structure infinie. En effet, par la loi de continuité, la transition $q_8 \xrightarrow{\pi} q_1$ peut être décomposée en une infinité d'étapes temporelles, dont le cumul est égal à π. Dans ce système, on peut remarquer un état puits (i.e. un état qui n'admet aucun successeur), l'état $q5$ et une transition silencieuse $q_7 \xrightarrow{\epsilon} q_3$.

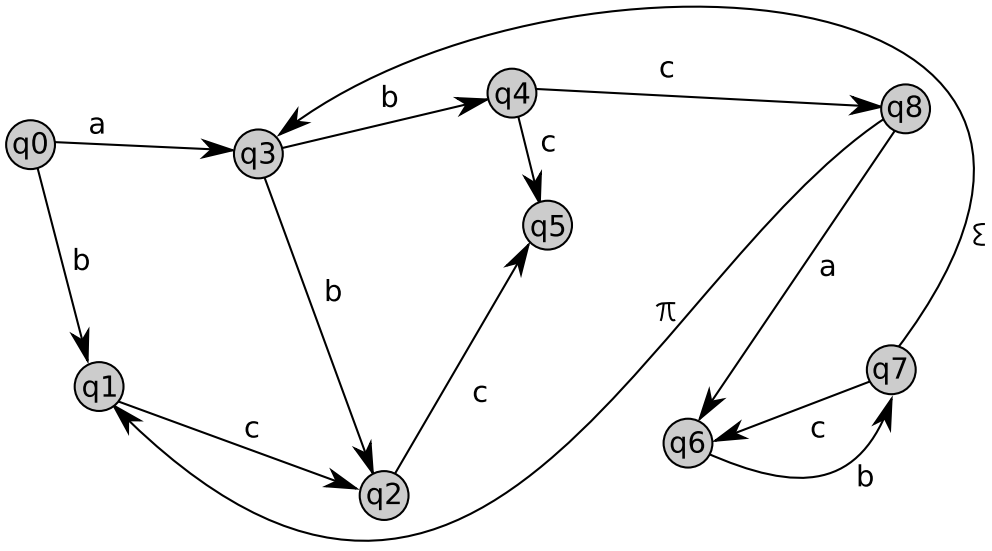


Fig. 2.2 – Un système de transitions temporisé

2.3 Logique Temporelle à temps Linéaire

Nous rentrons avec cette section dans l'*après-modélisation*. Que fait-on une fois le modèle conçu ? Nous sommes capables de donner le comportement d'un modèle sous la forme d'une relation de transitions, mais comment en extrait-on des informations ?

Les informations que l'on souhaite extraire sont les *propriétés* du modèle (ou du comportement du modèle). Une propriété est donnée par la spécification d'une *formule logique*, exprimant (lorsque la formule est valide) un comportement inclus dans celui du modèle sur lequel on veut avoir des informations. Les propriétés que nous utilisons dans cette thèse sont *qualitatives* c'est-à-dire caractérisées par le fait que l'on ne puisse répondre que par *oui* ou par *non* à la question : "la propriété est-elle validée par le modèle ?". La formule (même si elle utilise des informations quantitatives) dénote une qualité présente (ou non) dans le modèle.

Les concepts et notations des logiques modales utilisées en informatique sont issus de travaux en philosophie traitant des modalités comme la possibilité et l'obligation [HC68].

Nous allons présenter en détail une de ces logiques, la logique temporelle à temps linéaire (*LTL*), qui se révèle très utile pour la spécification de propriétés, permettant de raisonner sur l'ensemble des séquences d'exécution du modèle. Il est par exemple possible d'exprimer qu'un évènement intervient avant un autre, ou que dans le futur (sans plus de précisions) un évènement surviendra. Le séquencement des actions est logique, c'est-à-dire qu'aucune quantification (de durée ou de distance, par exemple) n'est spécifiable, contrairement à d'autres logiques permettant de quantifier temporellement l'enchaînement des actions, comme la logique *MTL* [Koy90]. Elle permet également de décrire qu'un modèle fait effectivement quelque chose (*propriété de vivacité*, ou *liveness*) et des propriétés plus simples à vérifier comme la détection d'erreurs, qui est ce que l'on appelle une *propriété de sûreté* (*safety*) : le système, quoi qu'il fasse, n'atteint pas un état erroné.

Définition 3 *Syntaxe de* LTL

Soit PA un ensemble de propositions atomiques et $p \in PA$. Les formules LTL *sont obtenues par la grammaire suivante :*

$$\Phi ::= p \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \mathcal{U} \Phi_2 \mid \mathcal{N}\Phi \tag{2.1}$$

Une formule *LTL* est vrai pour un modèle si et seulement si elle est vraie pour l'ensemble des exécutions de ce modèle : comme nous allons le voir dans sa sémantique, une formule est validée de manière récursive à partir de l'état initial, en parcourant un à un tous les chemins d'exécutions.

Définition 4 *Sémantique de* LTL

Soit PA un ensemble de propositions atomiques et p une de ces propositions ; $\sigma = s_0 s_1 ...$ une séquence infinie d'états ; $\nu : S \to 2^{PA}$ une fonction d'étiquetage, $\nu(s_i)$ définissant donc l'ensemble des propositions atomiques vérifiées par l'état s_i. $\sigma^i \models \Phi$ signifie que la formule Φ est satisfaite par σ à la position i.

$\sigma^i \models p$ *ssi* $p \in \nu(s_i)$
$\sigma^i \models \neg\Phi$ *ssi* $\sigma^i \not\models \Phi$
$\sigma^i \models \Phi_1 \vee \Phi_2$ *ssi* $\sigma^i \models \Phi_1$ *ou* $\sigma^i \models \Phi_2$
$\sigma^i \models \Phi_1 \mathcal{U} \Phi_2$ *ssi* $\exists j \geqslant i$ *tel que* $\sigma^j \models \Phi_2$ *et* $(\forall k)(i \leqslant k < j \Rightarrow \sigma^k \models \Phi_1)$
$\sigma^i \models \mathcal{N}\Phi$ *ssi* $\sigma^{i+1} \models \Phi$

Les autres opérateurs usuels sont définis en fonction de ces quatre fondamentaux :
$\wedge$ (et) $a \wedge b = \neg(\neg a \vee \neg b)$
$\Rightarrow$ (implique) $a \Rightarrow b = \neg a \vee b$
$\Diamond$ (finalement) $\Diamond a = \mathbf{true} \mathcal{U} a$
$\Box$ (toujours) $\Box a = \neg\Diamond(\neg a)$

La définition ci-dessus n'accepte que des séquences infinies. Pour pouvoir appliquer cette définition sur les séquences finies, il suffit de répéter infiniment les états ne possédant aucun successeurs, en les reliant par des transitions silencieuses (i.e. étiquetées par ϵ, la lettre neutre pour la concaténation).

La sémantique de *LTL* suppose que les états sont étiquetés. Dans le cas des systèmes de transition temporisés ce sont les transitions qui sont étiquetées. Pour se ramener à une structure utilisable pour *LTL*, il convient donc soit de déporter l'information sur les états, soit d'utiliser *State/Event LTL* [CCO+04], une variante supportant à la fois les propositions atomiques sur les états et les transitions.

Il est également possible de repérer un état puits (i.e. un état ne possédant pas de successeurs) en étiquetant cet état par une étiquette spéciale *puits*. La détection d'un état puits dans le comportement d'un modèle se caractérise par l'invalidation de la formule $\Box\neg puits$ qui signifie «pour toutes les séquences, il est toujours vrai que l'état n'est pas étiqueté par puits» : lorsque cette formule est fausse, cela indique qu'il existe des séquences pour lesquelles un état est marqué par *puits*. Utiliser $\Diamond puits$ n'est pas correct pour la détection d'états puits, car cette formule signifie que *toutes* les exécutions se terminent par cet état. Il est par contre possible d'utiliser la formule $\neg\Diamond puits$ «pour toutes les séquences, il n'est pas possible d'arriver dans un état marqué puits» qui est équivalente à $\Box\neg puits$.

En reprenant le *TTS* donné en figure 2.2, on peut observer que la formule $\Box\neg puits$ est fausse, car il existe un état puits : *q5*; c'est également le cas pour la formule $\Diamond puits$, car ce *TTS* possède des séquences infinies.

Pour conclure cette rapide introduction de la logique *LTL*, la formule suivante est typique des formes utilisées pour la vérification des systèmes réactifs :

$$\Box(c \Rightarrow \Diamond a) \tag{2.2}$$

Elle permet de savoir si à tout évènement c, on peut trouver (pour toutes les séquences possibles) un évènement a intervenant dans son futur, auquel cas, a peut être considéré comme la réponse à une requête c. Dans le système de la figure 2.2, cette formule est fausse car il existe une séquence infinie qui l'invalide : $q_0.q_3.q_4.q_8.(q_6.q_7)^\omega$.

2.4 *Model checking* de *LTL* à l'aide des automates de Büchi

Nous savons donner (quoi que de manière très peu pratique pour l'instant) le comportement d'un modèle à l'aide des *TTS* ; nous pouvons également spécifier des propriétés de ce modèle par des formules de la logique *LTL*. Le lien entre les deux s'établit en transformant les formules *LTL* en automates de Büchi composables avec le modèle du comportement. Un automate de Büchi est différent d'un automate classique car il permet de définir un langage de mots infinis (contre un langage de mots finis pour les automates classiques). Un mot infini est aussi appelée *trace*, mais nous préférerons le terme *exécution* dans la suite.

En exprimant un modèle, nous donnons l'ensemble des exécutions du système que l'on souhaite analyser. En exprimant une propriété, nous donnons un ensemble d'exécutions que l'on souhaite retrouver dans le comportement du système. Un comportement, lorsqu'il est analysé par une logique de chemins comme *LTL*, est formalisable par un ensemble de séquences de transitions (ou *traces*, ou *exécutions*). A chaque transition est associée une lettre d'un alphabet Σ ; une séquence de transitions définit ainsi un mot, et l'ensemble des séquences donnant le comportement du modèle définit un langage L_Σ. Si le modèle est repéré par M, alors $L_\Sigma(M)$ donne son comportement, où Σ est l'ensemble des actions du modèle. De la même façon, une propriété P admet un langage $L_\Sigma(P)$. Ainsi, vérifier qu'une propriété est valide pour un modèle revient à vérifier l'inclusion du comportement du modèle dans celui de la propriété :

$$L_\Sigma(M) \subseteq L_\Sigma(P) \tag{2.3}$$

Deux méthodes sont alors possibles pour vérifier cette inclusion.

- Soit directement par $L_\Sigma(P) \cap L_\Sigma(M)$: si cette intersection est égale à $L_\Sigma(M)$, alors la propriété est valide, elle ne l'est plus dès qu'une exécution du modèle n'est pas entièrement englobée par le comportement de la propriété.
- Soit en niant la propriété : $\overline{L_\Sigma(P)} \cap L_\Sigma(M)$. Si l'intersection est vide alors la formule spécifiant la propriété P est vraie puisqu'il n'y a aucune exécution satisfaisant la propriété $\overline{L_\Sigma(P)}$.

La première méthode est plus compliquée à mettre en place puisqu'il faut pouvoir tester l'égalité entre l'intersection et le modèle lui-même. Etape qui n'est pas nécessaire dans le deuxième cas, puisqu'il est suffisant de tester la vacuité du langage.

Mis à part les problèmes de faisabilité et de complexité de l'intersection, de la complémentation, du test de langage vide et de l'égalité de comportement, qui sont les opérations potentiellement utilisées par les deux méthodes, la deuxième méthode permet de connaître directement les traces qui falsifient la propriété, desquelles il est possible d'extraire un contre-exemple, ce qui n'est pas possible avec la première méthode : seules les traces respectant la formule étant gardées.

Notre propos n'étant pas de donner les détails permettant d'accomplir ces différentes opérations, nous en resterons à ce niveau de détail. Pour plus de précision, nous recommandons la lecture de [CGP00] et de [DL07].

Le processus de vérification est donné par la figure 2.3

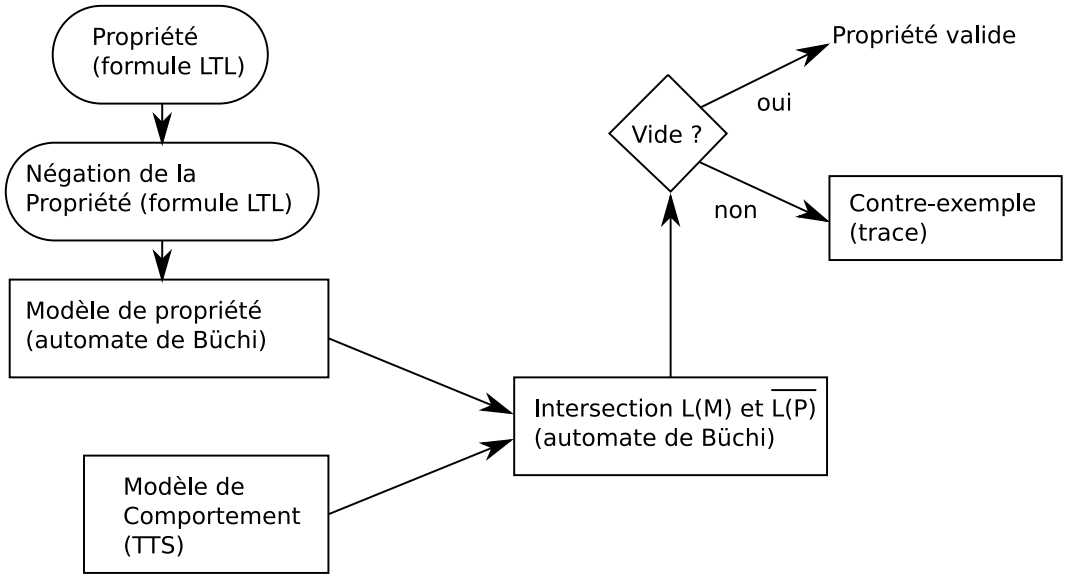


Fig. 2.3 – Vérification de propriété LTL

2.5 Décidabilité de l'accessibilité des états

L'indécidabilité : une limite théorique de la vérification automatique

Un des problèmes les plus handicapant du *model checking* est la limite théorique de l'indécidabilité de l'accessibilité d'un état pour les formalismes capables de simuler une machine de Turing. Le résultat fondamental de Turing est que le test de l'arrêt d'une machine de Turing est indécidable : il n'existe pas de programme exécutable sur une machine de Turing capable d'implémenter ce test. L'accessibilité d'un état est une généralisation du test de l'arrêt d'une machine de Turing : tester si un programme se termine (la machine s'arrête) revient à tester si l'état dans lequel la machine s'arrête est accessible.

Pour savoir si un état d'un système est accessible, il suffit de construire le comportement du système : si un état est présent dans la représentation du comportement, cela signifie qu'il est accessible. Le caractère fini d'un comportement implique donc la décidabilité de l'accessibilité de ses états. De plus, lorsque l'accessibilité des états d'un comportement est décidable alors il existe un algorithme construisant une structure *finie* représentant le comportement, qui lui peut être fini. Il faut bien faire attention : un comportement infini n'implique *pas* l'indécidabilité de l'accessibilité de ses états.

Puisqu'une représentation finie du comportement est nécessaire pour appliquer la technique de vérification par force brute (*brute force*) qu'est le *model checking*, l'indécidabilité est donc une limite

dont il nous faut préciser les contours, et ce afin de connaître les systèmes temps réel qu'il est possible de vérifier sans tomber dans cette problématique.

Des résultats de décidabilité pour les formalismes hybrides

Comme nous allons le voir plus en détail au chapitre 4, la correction d'un système temps réel dépend du respect de ses échéances. La dynamique de ces systèmes implique quasiment systématiquement l'utilisation d'un ordonnanceur capable de déterminer les tâches les plus prioritaires à exécuter. Il est possible que cet ordonnanceur soit autorisé à suspendre une tâche en cours d'exécution pour en laisser une autre, plus prioritaire, s'exécuter. La tâche suspendue ne l'est que temporairement et son échéance n'en est pas pour autant retardée. Ainsi la spécification de tels systèmes doit tenir le compte des horloges suspendues et de celles qui ne le sont pas. Les horloges sont des variables réelles sur lesquelles on peut appliquer soit une fonction d'incrémentation synchrone au temps du système, soit les laisser telles quelles. Ceci nous fait rentrer dans le domaine des systèmes hybrides, dont les résultats en termes de décidabilité ne sont pas en la faveur d'une technique automatique.

Un système hybride est caractérisé par une dynamique à la fois discrète et continue. L'un des formalismes les plus connus sont les *automates hybrides* formés à partir d'un automate classique sur lequel sont ajoutées des informations et des contraintes de progrès de variables continues.

En limitant l'expression des fonctions d'évolution des variables continues, on obtient différentes classes d'automates hybrides. Voici un court aperçu des résultats de décidabilité liés aux automates hybrides en fonctions des limitations imposées aux fonctions d'évolutions.

Automates temporisés [ACD90]. Cette classe est l'une des plus restreintes des automates hybrides. La fonction d'évolution des horloges est limitée à une dérivée constante égale pour toutes les horloges, qui sont les seules variables continues du formalisme. L'accessibilité est décidable [AD94].

Les automates hybrides initialisés. Les classes d'automates qui vont suivre possèdent la caractéristique d'être initialisées : lorsque l'évolution des variables continues change (par une transition discrète), les variables sont réinitialisées (à une valeur arbitraire).

Automates initialisés à multi-vitesses [ACHH93],[NOSY93] [HKPV95]. Cette classe autorise des évolutions différentes pour les variables continues, mais toutes ces évolutions sont des dérivées première (vitesse) fixe (une valeur de vitesse possible). L'accessibilité pour cette classe est décidable. Les automates de cette classes sont tous traduisibles en automates temporisés temporellement bisimilaires (la notion de bisimulation est définie dans la section suivante).

Automates initialisés rectangulaires [HKPV95]. Cette classe est une généralisation de la précédente car elle autorise un intervalle de vitesse au lieu d'une seule valeur définissant la vitesse d'évolution des variables continues.

Puisque les automates sont initialisés, il n'est pas possible de spécifier que l'évolution d'une variable (horloge) s'interrompt et reprend normalement après un délai arbitraire. Pour cela il faut autoriser la condition d'initialisation.

Un résultat de [HKPV95], montre qu'en enlevant la condition d'initialisation, mais en restreignant avec : l'invariant est constant, non strict et borné ; lors du tir d'une transition discrète les variables peuvent soit être remises à zéro, soit conserver leur valeur ; la garde est non stricte et bornée ; au plus une seule variable n'est pas une horloge (la vitesse d'une horloge est constante et égale à 1) ; l'accessibilité pour cette restriction drastique des automates hybrides non nécessairement initialisable est *indécidable* si la variable qui n'est pas une horloge admet deux vitesses : 1 (similaire à une horloge) et $x \in \mathbb{Q} \setminus \{1\}$.

Les horloges marquant la préemption des tâches doivent posséder deux vitesses : 0 et 1, ce qui rentre dans ce résultat l'indécidabilité de l'accessibilité des états.

Ceci est vrai sur tous les formalismes possédant une caractéristique équivalente et notamment pour les automates temporisés à chronomètres [CL00], les réseaux de Petri ordonnanceurs [RD02], les réseaux de Petri préemptifs [BFSV03], les réseaux de Petri temporel à hyper-arcs inhibiteurs [RL04] et les réseaux de Petri temporels à chronomètres [BLRV05].

Ce problème l'indécidabilité n'est pas réservé à l'expression des variables continues, car pour le

formalisme des réseaux de Petri temporels, que nous allons présenter dans la suite, l'accessibilité des états est indécidable lorsque le réseau de Petri sous-jacent (sans informations temporelles) est non-borné. Nous verrons cela plus en détail dans la suite de ce chapitre.

2.6 Bisimulation

Pouvoir raisonner sur des systèmes modélisés à des niveaux d'abstractions ou de raffinements différents exige souvent de savoir les comparer. L'équivalence langage ou l'isomorphisme sont utilisables pour comparer deux relations de transition. Il est par exemple possible d'employer l'équivalence langage lors d'un processus de modélisation par raffinements successifs, le raffinement n'étant acceptable que s'il est équivalent à son abstraction. L'isomorphisme n'est pas adapté pour ce genre de méthodologie, car alors la marge de manœuvre laissée à une abstraction ou à un raffinement est restreinte à la seule possibilité de renommer les termes utilisés dans le modèle. L'équivalence langage est par contre trop faible pour vérifier que les choix (les branchements) au cours d'une exécution sont possibles aux même endroits dans les deux relations de transitions comparées. Elle ne préserve pas non plus la notion de blocages. Dans ce cas, l'isomorphisme est également une équivalence peu intéressante, car même si les branchements sont bien préservés, aucune liberté n'est permise : les deux comportements doivent être identiques au nom près.

La notion de bisimulation est une notion d'équivalence intermédiaire [San04] très utile pour la comparaison de comportements et répondant à la problématique par raffinements successifs. De manière générale, elle permet de vérifier que deux comportements sont «similaires» : si un système fait une action, alors l'autre système fait également cette action et réciproquement. De plus, il est également possible de se servir d'une variante «faible» de cette abstraction, ne prenant en compte que les actions non silencieuses (observables), pour déterminer si une propriété, codée sous la forme d'un automate, est valide pour un comportement (dont on aura rendue non observables les actions n'apparaissant pas dans la propriété) : si celui-ci est faiblement bisimilaire à la propriété, cela signifie que la propriété est valide pour ce comportement.

Un langage formel pouvant être défini par l'ensemble de comportements qu'il peut exprimer, cette équivalence peut également servir pour comparer deux formalismes.

Il est possible d'étendre cette équivalence pour l'adapter à la comparaison de *TTS*. On obtient ainsi la bisimulation temporisée [Yi90], notion plus fine que la bisimulation, particulièrement adaptée pour comparer deux formalismes utilisant une notion de temps.

Définition 5 Bisimulation temporisée *: Soient* $S1 = \langle Q_1, q_1^0, \Sigma_1^\epsilon, \rightarrow_1 \rangle$ *et* $S2 = \langle Q_2, q_2^0, \Sigma_2^\epsilon \ \rightarrow_2 \rangle$ *deux systèmes de transitions temporisés et* $\sim \subseteq Q_1 \times Q_2$*. Alors* S_1 *et* S_2 *sont* fortement temporellement bisimilaires *ssi* $q_1^0 \sim q_2^0$ *et, quand* $q_1 \sim q_2$ *et* $a \in \Sigma_1^\epsilon \cup \Sigma_2^\epsilon \cup \mathbb{R}^+$ *:*

(1) $q_1 \xrightarrow{a}_1 q_1' \Rightarrow (\exists q_2')(q_2 \xrightarrow{a}_2 q_2' \wedge q_1' \sim q_2')$

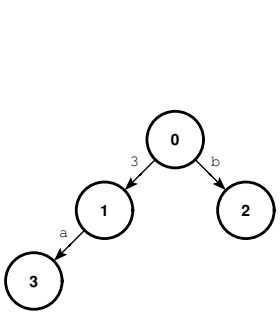
(2) $q_2 \xrightarrow{a}_2 q_2' \Rightarrow (\exists q_1')(q_1 \xrightarrow{a}_1 q_1' \wedge q_1' \sim q_2')$

En pratique, la bisimilarité temporisée forte est souvent trop contraignante. Une relation d'équivalence plus grossière, cachant les transitions silencieuses, est obtenue à partir de la relation $\stackrel{a}{\Rightarrow}$, définie depuis $\stackrel{a}{\rightarrow}$ comme suit ($a \in \Sigma \cup \mathbb{R}^+, d \in \mathbb{R}^+$) :

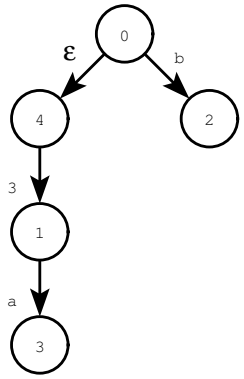
$$\frac{q \xrightarrow{a} q'}{q \stackrel{a}{\Rightarrow} q'} \qquad \frac{q \stackrel{a}{\Rightarrow} q' \quad q' \xrightarrow{\epsilon} q''}{q \stackrel{a}{\Rightarrow} q''} \qquad \frac{q \xrightarrow{\epsilon} q' \quad q' \stackrel{a}{\Rightarrow} q''}{q \stackrel{a}{\Rightarrow} q''} \qquad \frac{q \stackrel{d}{\Rightarrow} q' \quad q' \stackrel{d'}{\Rightarrow} q''}{q \stackrel{d+d'}{\Longrightarrow} q''}$$

Deux systèmes de transitions temporisés seront dits *faiblement temporellement bisimilaires* lorsque les conditions (1) et (2) ci-dessus sont satisfaites, avec les relations $\xrightarrow{a}_i$ remplacées par $\stackrel{a}{\Rightarrow}_i$ ($i \in \{1, 2\}$). La bisimulation temporisée faible est notée $\sim^w$.

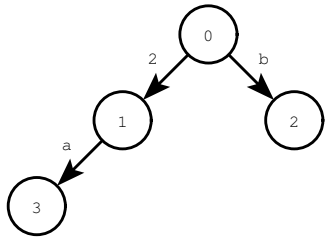
La version atemporelle s'obtient de la définition 5 en enlevant la possibilité à a d'être dans $\mathbb{R}^+$. La bisimulation atemporelle est notée $\sim^a$.



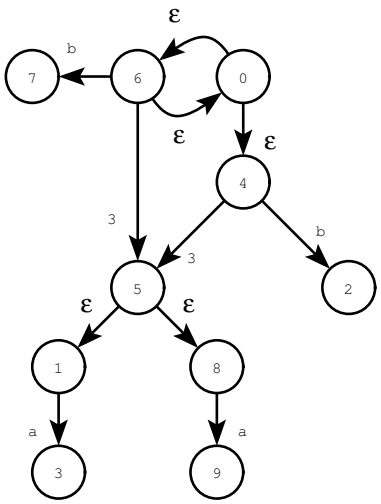
α



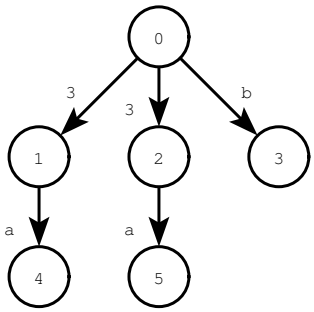
β
$\alpha \not\sim^w \beta$



γ
$\alpha \sim^a \gamma$
$\alpha \not\sim^w \gamma$



δ
$\alpha \sim^w \delta$



ϵ
$\alpha \sim \epsilon$

Le système de transition temporisé α n'est pas bisimilaire avec β, car l'état 4 de β doit être équivalent à 0 de α, or on ne peut pas faire de b à partir de l'état 4. Le TTS γ n'est qu'atemporellement bisimilaire puisque le délai n'est pas le bon. Le TTS ϵ est bien fortement bisimilaire à α, car les états 1 et 2 sont bisimilaires à l'état 1 de α et de ces états bisimilaires, on peut aller dans les états 4 à partir de 1 et 5 à partir de 2 par la même transition a et les états 4 et 5 sont bisimilaires à 3 dans α.

Finalement, δ est faiblement temporellement bisimilaire à α, car les états $0, 4$ et 6 sont équivalents à l'état 0 de α ; de ces états on peut toujours faire soit une action b soit attendre 3 unités de temps ; et les états atteints ont des états bisimilaires dans α : 7 et 2 sont bisimilaires à 2 dans α et 5 est bisimilaire à l'état 1 de α. Finalement de l'état 5, on ne peux faire que des transitions silencieuses menant aux états 1 et 8, également bisimilaires à l'état 1 de α car ils permettent tous deux de faire l'action a et les états atteint (4 et 5) sont bisimilaires à l'état 3 dans α.

2.7 Réseaux de Petri temporels

La définition algébrique des réseaux de Petri a été introduite dans la thèse de Carl Adam Petri en 1962 [Pet62] (1966 pour sa traduction anglaise "*communication with automata*" [Pet66]), les motivations de sa proposition étant que les réseaux de Petri représentaient plus fidèlement les phénomènes physique que le formalisme des automates. Le caractère d'adaptation au monde physique s'est propagé dans le monde de l'automatisme, mais pas seulement. En effet les réseaux de Petri ont la capacité in-

trinsèque de pouvoir représenter aisément des systèmes concurrents car ils supportent le parallélisme de manière native. L'informatique s'est donc très vite appropriée ce formalisme pour spécifier des protocoles de communications, etc.

A la différence des automates qui sont basés sur des graphes orientés, les réseaux de Petri sont basés sur des graphes orientés bipartites. Alors qu'un graphe «normal» est constitué d'un seul type de nœuds, un graphe bipartite est, lui, constitué de deux types de nœuds. Dans le cas des réseaux de Petri, ces deux types de nœuds sont les places et les transitions. Une place porte une information partielle de l'état du système tandis qu'une transition est capable de modifier l'information portée par un sous-ensemble de places.

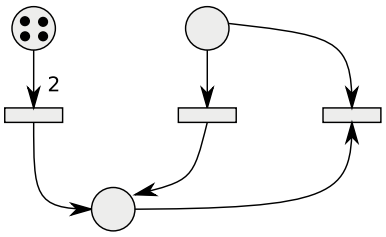


Fig. 2.4 – Un réseau de Petri est un graphe bipartite

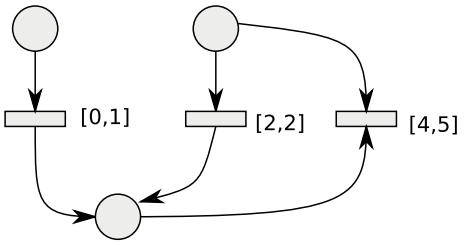


Fig. 2.5 – Un réseau de Petri temporel

Traditionnellement, les réseaux de Petri sont représentés sous leur forme graphique par des cercles pour les places et des rectangles (ou des traits) pour les transitions (fig. 2.4). Le marquage initial (voir définition ci-dessous) est donné par un nombre à l'intérieur ou des points lorsque le nombre est suffisamment petit. Les fonctions d'incidences (définition ci-dessous) avant et arrière sont représentées par des arcs pondérés et orientés entre une place et une transition.

Le temps peut être modélisé de manière discrète ou non. Tout modèle de temps discret présuppose un grain de temps minimal fixé *a priori*. Les modèles synchrones s'accommodent parfaitement de cette présupposition, puisqu'ils sont basés sur *l'hypothèse synchrone* : un système synchrone réagit suffisamment rapidement pour détecter tout changement de son environnement. Par contre, un modèle asynchrone ne suit pas forcément cette hypothèse : il est donc possible de rater des comportements erronés en discrétisant.

Le modèle discret (ou causal) du temps peut être modélisé par un réseau de Petri (typiquement une place pour garder la quantification et une transition *tick* pour dénoter la progression du temps, voir par exemple [Pom02]). Ce n'est par contre pas le cas pour le modèle dense du temps. Les systèmes temps réel qui nous intéressent dans cette thèse sont pris dans une acception la plus générale possible et ne respectent donc pas nécessairement l'hypothèse synchrone : un évènement peut intervenir à un instant arbitrairement proche d'un autre. Cela implique qu'il n'est pas possible de fixer un grain minimal de progression du temps et partant, qu'il est obligatoire de ne pas utiliser un modèle à temps discret si l'on souhaite capturer tous les comportements. Des modèles ont été proposés permettant d'étendre les réseaux de Petri en vue de gérer le temps de manière plus explicite : parmi ceux-là, nous nous intéresserons aux réseaux de Petri temporels de [Mer74].

Dans [Mer74], l'auteur étend les réseaux de Petri en associant deux valeurs aux transitions : une borne minimale a et une borne maximale b, toutes deux positives, représentant les instants entre lesquels la transition peut tirer (voir figure 2.5). Dès qu'une transition est sensibilisée ces deux valeurs se mettent à décroître de manière synchrone au temps et dès que a arrive à zéro, la transition devient tirable. Si la transition n'est pas tirée, le temps peut continuer à avancer et la deuxième borne b continue de décroître jusqu'à ce que sa valeur soit nulle, auquel cas, la transition est obligée de tirer ou d'être désensibilisée par une autre.

Les réseaux de Petri temporels se révèlent être très utiles pour la modélisation de protocoles de communications, pour lesquels une notion d'incertitude de la date d'arrivée des évènements est bienvenue ; ou plus généralement pour la modélisation de systèmes temps réel de par son aptitude à modéliser des *time out* (attentes temporisées).

Dans la suite de ce chapitre nous présentons le formalisme des réseaux de Petri temporels qui sera

notre base de travail tout au long de cette thèse : après sa définition, nous présentons sa sémantique, puis nous discutons de la décidabilité de l'accessibilité des états. Nous montrons ensuite une méthode de représentation du comportement des modèles des réseaux de Petri temporels, pour finir sur des extensions qui nous serons utiles dans les chapitres suivants.

2.7.1 Définitions

Définition 6 $\mathbf{I}^+$ *est l'ensemble des intervalles définis comme suit :*
- $\{x \in \mathbb{R} \mid (a,b) \in \mathbb{Q}^2 \wedge 0 \leqslant a < x < b\} =]a,b[$
- $\{x \in \mathbb{R} \mid (a,b) \in \mathbb{Q}^2 \wedge 0 \leqslant a \leqslant x < b\} = [a,b[$
- $\{x \in \mathbb{R} \mid (a,b) \in \mathbb{Q}^2 \wedge 0 \leqslant a < x \leqslant b\} =]a,b]$
- $\{x \in \mathbb{R} \mid (a,b) \in \mathbb{Q}^2 \wedge 0 \leqslant a \leqslant x \leqslant b\} = [a,b]$
- $\{x \in \mathbb{R} \mid a \in \mathbb{Q} \wedge 0 \leqslant a \leqslant x\} = [a,\infty[$
- $\{x \in \mathbb{R} \mid a \in \mathbb{Q} \wedge 0 \leqslant a < x\} =]a,\infty[$

Il est utile de pouvoir parler des bornes des intervalles. Les bornes d'un intervalle sont les réels qui l'encadre. Formellement : à tout intervalle $\{x \in \mathbb{R} \mid (a,b) \in \mathbb{Q}^2 \wedge 0 \leqslant a \prec x \prec b\}$, avec $\prec \in \{<, \leqslant\}$, ses bornes sont a et b, notées $\downarrow i = a$ et $\uparrow i = b$. Si l'intervalle est non-borné, il ne possède pas de borne maximale $\uparrow i = b$.

Les bornes peuvent ne pas appartenir à l'intervalle qu'elles caractérisent. Ceci peut poser un problème : en ne connaissant que l'information des bornes il n'est pas possible de retrouver l'intervalle à l'origine de ces bornes. Dans le but d'une réelle utilisation de l'information des bornes nous permettant de modulariser les définitions, nous étendons cette notion en rajoutant l'information d'appartenance d'une borne à son intervalle.

Définition 7 *Pour tout intervalle* $\{x \in \mathbb{R} \mid a \in \mathbb{Q} \wedge b \in \mathbb{Q} \cup \{\infty\} \wedge 0 \leqslant a \prec x \prec b\}$*, avec* $\prec \in \{<, \leqslant\}$*, ses bornes sont les couples de* $\mathbb{Q} \times \mathbb{B}$[1] *:*
- $\downarrow i = (a, \top)$ *si* $\prec = \leqslant$
- $\downarrow i = (a, \bot)$ *si* $\prec = <$*,*
- $\uparrow i = (b, \top)$ *si* $\prec = \leqslant$ *et*
- $\uparrow i = (b, \bot)$ *si* $\prec = <$*.*

Cette deuxième définition sera plus généralement préférée par la suite. Puisque une borne n'est plus un nombre, il est nécessaire de redéfinir l'opération de comparaison :

Définition 8 *Soient* $x \in \mathbb{R}$ *et* $\alpha = (a, s) \in Q \times \mathbb{B}$*,*
- $x \leqslant \alpha$ *est défini par*
  - *si* $s = \bot$ *alors* $x < a$
  - *si* $s = \top$ *alors* $x \leqslant a$
- *et* $x < \alpha$ *est défini par* $x < a$

Nous pouvons à présent donner la définition des réseaux de Petri temporels étiquetés :

Définition 9 *Un réseau de Petri temporel étiqueté est un n-uplet* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s, \Sigma^\epsilon, \mathcal{L}\rangle$*, où :*
- P *est l'ensemble des* places
- T *est l'ensemble des* transitions
- $P \cap T = \emptyset$
- $\mathbf{Pre} : T \times P \to \mathbb{N}$ *est la* fonction d'incidence avant*,*
- $\mathbf{Post} : T \times P \to \mathbb{N}$ *est la* fonction d'incidence arrière*,*
- m_0 *est le* marquage initial*,*
- $I^s : T \to \mathbf{I}^+$ *est la* fonction intervalle statique*,*

[1] où $\mathbb{B}$ dénote l'ensemble des booléens et $\top$ et $\bot$ sont ses deux éléments vrai et faux

– Σ *est un alphabet fini d'*actions
– $\mathcal{L} : T \times \Sigma$ *est la* fonction d'étiquetage *des transitions*

Il est à noter que l'étiquetage n'est pas une donnée essentielle pour la définition des réseaux de Petri, nous en constaterons néanmoins l'utilité tout au long de ce manuscrit de thèse.

Bien que les réseaux de Petri possèdent évidemment une définition bien distincte (sans I_s), nous préférons utiliser celle des réseaux de Petri temporels pour définir les réseaux de Petri (atemporels), montrant ainsi que ces derniers peuvent être considérés comme un cas particulier des premiers.

Définition 10 *Un réseau de Petri temporel étiqueté* $N = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s, \Sigma^\epsilon, \mathcal{L} \rangle$ *est un réseau de Petri étiqueté ssi* I_s *respecte* $(\forall t \in T)(I_s(t) = [0, \infty[)$.

A tout *TPN* on peut associer son réseau atemporel sous-jacent en transformant tous les intervalles de tir en $[0, \infty[$. Le réseau de la figure 2.4 est le réseau atemporel sous-jacent de celui de la figure 2.5.

Un multi-ensemble est un couple (A, f), où A est un ensemble, appelé ensemble support et $f : A \rightarrow \mathbb{N}$ une fonction. Formellement, un marquage noté m est un multi-ensemble (P, m) ayant pour support l'ensemble des places P. Ainsi, $m(p)$ donne le nombre de jetons de la place p dans le marquage m (autrement dit, le nombre d'occurrences $m(p)$ de la place $p \in P$ dans le multi-ensemble (P, m)).

On peut définir l'opération d'addition sur les marquages de la manière suivante : soit l, m, n trois marquages et $l = m + n$, alors $(\forall p \in P)(l(p) = m(p) + n(p))$. De même, on peut définir l'opérateur de comparaison $\prec \in \{<, \leqslant, =, \geqslant, >\}$ par $m \prec n \Leftrightarrow (\forall p \in P)(m(p) \prec n(p))$.

Pour chaque marquage m, on peut définir l'ensemble $\mathcal{E}n(m) = \{t \in T | m \geqslant \mathbf{Pre}(t)\}$. C'est l'ensemble des transitions *sensibilisées* par le marquage m.

La fonction d'incidence avant (resp. arrière) appliquée à une transition donne ce que l'on appelle les *préconditions* (resp. *postconditions*) de la transition.

Exemple

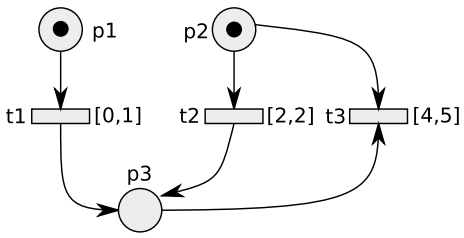


Fig. 2.6 – Un réseau de Petri temporel

Le réseau de la figure 2.6 est constitué de trois places *p1*, *p2* et *p3*. Les places *p1* et *p2* sont initialement marquées et le marquage initial $\mathcal{M}_0$ est tel que $\mathcal{M}_0(p1) = \mathcal{M}_0(p2) = 1$ et $\mathcal{M}_0(p3) = 0$. Les transitions sont *t1*, *t2* et *t3*. *t1* admet la place *p1* pour précondition et la place *p3* pour postcondition. La transition *t3* n'admet pas de postconditions mais possède *p2* et *p3* comme précondition. A *t1* est associé l'intervalle de tir $I_s(t1) = [0, 1]$, $I_s(t2) = [2, 2]$ pour *t2* et $I_s(t3) = [4, 5]$ pour *t3*. Nous verrons dans la section suivante, décrivant la sémantique, comment ces informations évoluent dynamiquement.

2.7.2 Sémantique

Traditionnellement, la sémantique des langages formels est exprimée à l'aide des concepts de transitions et d'états, une transition reliant deux états. Un tel ensemble de transitions est typiquement une relation, appelée la *relation de transition*, que l'on représente souvent sous la forme d'un graphe orienté.

Avant de donner l'ensemble des transitions, nous allons définir les éléments liés par ces transitions : les états.

Définition 11 Etat

Un état *d'un réseau de Petri temporel est donné par un couple (m, I), où m est un marquage et $I : T \to \mathbf{I}^+$ une fonction associant un intervalle à chacune des transitions sensibilisées.*

La sémantique des réseaux de Petri temporels *TPN* est donnée par sa traduction en un système de transitions temporisé.

Définition 12 *Sémantique d'un réseau de Petri temporel*

L'état initial est (m_0, I_0), où m_0 est le marquage initial du réseau et $I_0 : \mathcal{E}n(m) \to \mathbf{I}^+$ est défini par $(\forall t \in \mathcal{E}n(m))(I_0(t) = I_s(t))$.

Pour tout état (m, I) accessible depuis l'état initial, on accède à un état (m', I') par une transition discrète t, étiquetée par une lettre $\mathcal{L}(t)$ de l'alphabet Σ^ϵ ou par une transition temporisée continue (étiquetée par un réel x) :

$(m, I) \xrightarrow{\mathcal{L}(t) \in \Sigma^\epsilon} (m', I')$ *ssi*

1) $\mathbf{Pre}(t) \leqslant m$

2) $0 \in I(t)$

3) $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$

4) $(\forall k \in T \setminus \{t\})(\mathbf{Pre}(k) \leqslant m - \mathbf{Pre}(t) \Rightarrow I'(k) = I(k))$

5) $(\forall k \in T)(\mathbf{Pre}(k) \leqslant m' \Rightarrow I'(k) = I_s(k))$

$(m, I) \xrightarrow{x \in \mathbb{R}} (m, I')$ *ssi*

6) $(\forall t \in \mathcal{E}n(m))(x \leqslant \uparrow I(t))$

7) $(\forall t \in \mathcal{E}n(m))(I'(t) = I(t) \stackrel{\bullet}{-} x)$

Il y a deux conditions au tir de la transition t : la première est la sensibilisation de la transition (on dit que la transition est *sensibilisée* et la seconde est d'avoir franchi la borne minimale de tir. Si elle respecte ces deux conditions, une transition est dite *tirable*. Ensuite viennent les modifications du marquage et des intervalles de tir en conséquence du tir de la transition t. Le nouveau marquage est obtenu (c.f. règle 3) en retirant les jetons des places en préconditions et en ajoutant des jetons dans les places en postconditions. Les transitions persistantes (qui ne sont pas désensibilisées lors du tir de t) gardent leur intervalle de tir intact (règle 4). L'intervalle de tir des transitions nouvellement sensibilisées est leur intervalle statique de tir (règle 5).

En ce qui concerne la progression du temps, il ne peut aller au-delà des bornes supérieures des intervalles de tir des transitions sensibilisées (règle 6). Pour rappel, $\mathcal{E}n(m)$ est l'ensemble des transitions sensibilisées (*enabled* en anglais) par m. En conséquence de cet écoulement de temps, l'intervalle de tir des transitions sensibilisées est décalé de x unités de temps vers son origine (règle 7). Le décalage est effectué grâce à l'opération $\stackrel{\bullet}{-}$:

$$\stackrel{\bullet}{-} : \mathbf{I}^+ \times \mathbb{R} \to \mathbf{I}^+ \text{ est défini par : } Y \stackrel{\bullet}{-} x = \{a - x \mid a \in Y \wedge a \geqslant x)\} \tag{2.4}$$

Cette sémantique donne généralement une relation de transition infinie puisque d'un état à un autre il peut y avoir un nombre arbitrairement grand de transitions continues. Un exemple de comportement infini est donné sur la figure 2.7, avec un réseau basique comprenant une transition tirable après une unité de temps.

Les façons d'arriver à l'échéance de la transition sont infinies, aussi bien en branchement (ce que nous n'avons pas représenté sur la figure) qu'en nombre d'états. Les seuls états qui peuvent servir de référence, car ils sont obligatoirement franchis, sont l'état initial et les états reliés par le tir de la transition t.

Exemple

La figure 2.8 donne l'évolution de l'exemple utilisé dans la section précédente, selon la sémantique de la définition 12. Initialement, les transitions *t1* et *t2* sont sensibilisées. Comme $0 \in I(t) = [0, 1]$, alors *t1* est initialement tirable, mais *t2* ne l'est pas et doit attendre deux unités de temps pour cela.

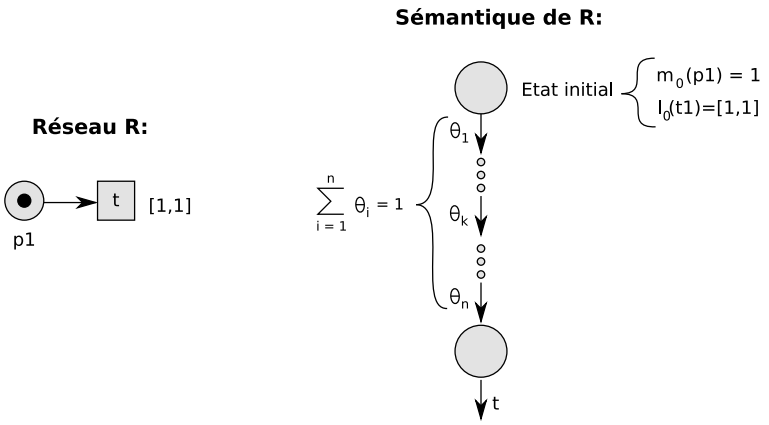


Fig. 2.7 – Le *TTS* d'un *TPN* est généralement infini

Le temps peut avancer d'une unité de temps (ou de toute valeur comprise entre 0 et 1), mais puisque le domaine de tir de *t1* est borné à 1, cela signifie que tant que cette transition est sensibilisée, le temps ne peut progresser au delà de 1. Soit $a \in [0,1]$ une valeur quelconque de progression du temps. Après cette progression, la transition *t1* est tirée, ce qui a pour effet de désensibiliser *t1* et de sensibiliser *t3*. A nouveau le temps peut progresser,mais pas au delà des bornes supérieures des intervalles de tir des transitions sensibilisées. Ceci implique que la progression b est telle que $b \leqslant 2 - a$ et donc que seule *t2* peut devenir tirable puisque $2 - a \leqslant 2$ et $2 < 4$. Seul le tir de *t2* est donc possible après que $a + b$ unités de temps se soient écoulés depuis l'état initial. Le tir de *t2* désensibilise aussi bien *t2* que *t3* et le système n'a donc plus que la possibilité de faire évoluer le temps sans qu'aucune transition ne devienne tirable.

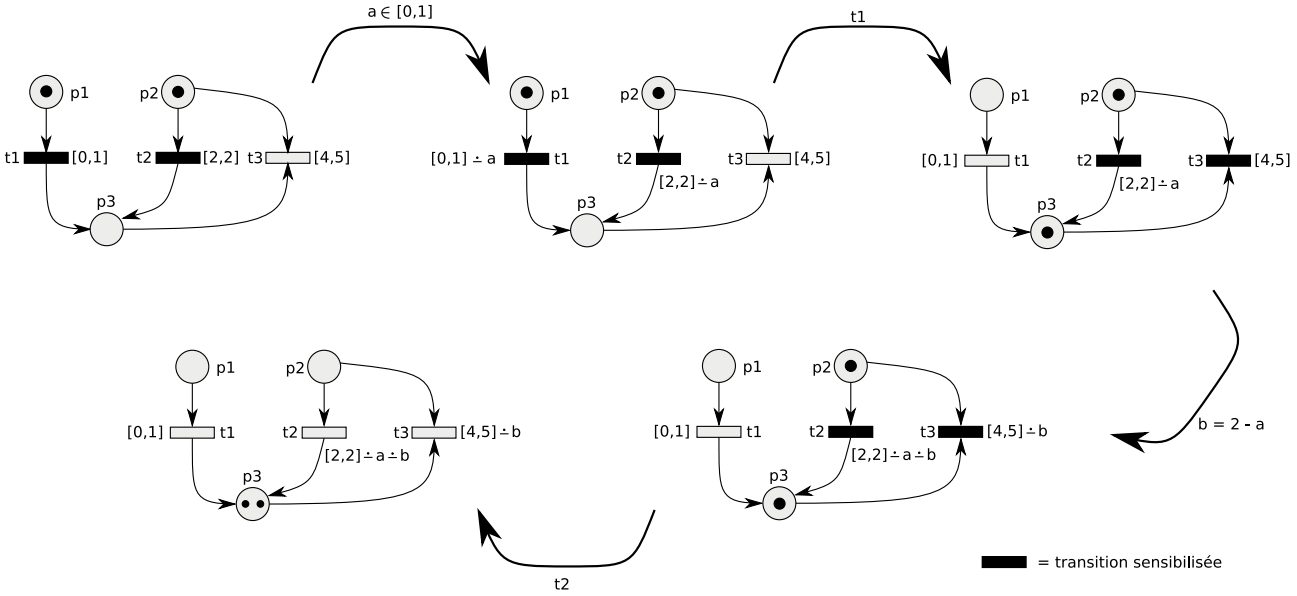


Fig. 2.8 – Evolution du Tpn exemple

Remarque

En toute rigueur, la sémantique ne précise pas le caractère obligatoire du tir d'une transition lorsque celle-ci admet un intervalle non borné. Il est seulement dit qu'il est possible d'attendre un temps arbitrairement grand avant de tirer, mais il n'est pas précisé si oui ou non ce délai peut être infini, c'est-à-dire qu'il n'est nulle part précisé si le cumul des délais peut être égal à l'infini. Il existe donc (au moins) deux interprétations : soit considérer que l'on peut attendre une infinité de temps et

ainsi autoriser à ne pas tirer la transition, soit considérer que l'on peut attendre un temps quelconque *avant* le tir et ainsi obligatoirement tirer celle-ci. Les deux interprétations ne sont pas équivalentes, car l'une autorise une trace maximale uniquement constituée de délais, alors que l'autre ne l'autorise pas, ce qui implique qu'elles ne préservent pas les même formules *LTL*.

Cela a deux conséquences : la première est qu'un réseau obtenu par la définition 10 n'est rigoureusement équivalent à sa version purement atemporelle qu'à la condition d'utiliser l'interprétation n'autorisant pas un cumul de délais infini. En effet, ce cumul infini, ou *divergence temporelle*, ne fait pas partie de la sémantique des réseaux de Petri : lorsqu'il y a au moins une transition tirable à partir d'un état, tout état successeur est forcément atteint par le tir de l'une des transitions sensibilisées. Une caractérisation plus rigoureuse des réseaux de Petri à partir des *TPN*, dans cette interprétation, serait que chaque transition doive porter l'intervalle $[0, 0]$, plutôt que $[0, \infty[$.

La deuxième conséquence concerne la construction du graphe de classe : selon l'une ou l'autre des interprétations, une transition silencieuse δ (δ peut être différent de ϵ pour plus de précision sur sa nature) devra être rajoutée ou non. Dans le cas de l'interprétation autorisant la divergence temporelle, lorsqu'une classe c admet un marquage tel que toutes les transitions sensibilisées sont temporellement non bornées, alors $c \xrightarrow{\delta} c$. Cette transition n'est jamais rajoutée dans l'autre interprétation.

2.7.3 Décidabilité

Nous présenterons principalement dans cette section un résultat concernant l'indécidabilité de l'arrêt de la construction du comportement des réseaux de Petri temporels, initialement prouvé dans [JLL77].

Définition 13 réseau borné

Un réseau de Petri (temporel ou non) est borné, ssi il existe un entier naturel k tel que pour tout marquage m accessible par le réseau à partir du marquage initial, et pour toute place p de ce réseau, on a $m(p) \leqslant k$. On dit alors que le réseau est $k-$borné. Un réseau $1-$borné est aussi appelé sauf.

Un réseau de Petri (atemporel) $k-$borné admet un comportement fini : le nombre de places est fini et le nombre total de marquages possibles est borné puisque chaque place admet une borne. Le nombre de transitions étant lui aussi fini, la relation de transition donnant son comportement est forcément finie.

Un réseau de Petri (quelconque) qui n'est pas $k-$borné admet forcément un nombre de marquages infini et par conséquent une relation de transition infinie.

Il est possible de décider si un réseau de Petri est $k-$borné, (en repérant les transitions qui augmentent strictement un marquage au cours de la construction du comportement, par exemple), par conséquent la possibilité de construction du comportement d'un réseau de Petri atemporel est décidable [KM69], [Fin93].

Si rajouter des intervalles de tir aux réseaux de Petri entraîne la création d'une infinité d'états, conséquence du nombre infini de transitions continues, ce rajout n'entraîne aucune addition de comportements discrets qui n'auraient pas eu lieu dans le réseau atemporel sous-jacent. C'est même le contraire qu'il se passe : les intervalles de tir étant des contraintes *supplémentaires* pour le tir des transitions, leur utilisation implique une réduction ou une conservation, mais pas une augmentation du nombre de transitions discrètes par rapport à celui du réseau atemporel sous-jacent. Ceci est particulièrement visible lorsqu'on considère les réseaux de Petri comme des réseaux de Petri temporels avec des intervalles de tirs triviaux ($[0, \infty[$). En conséquence, la construction du comportement d'un réseau temporel $k-$borné termine forcément.

Il existe par contre des réseaux temporels dont la partie atemporelle sous-jacente est non-bornée et qui restent malgré tout bornés. Ceci est dû aux contraintes temporelles qui limitent la fréquence de tir des transitions de manière à garder l'ensemble des marquages bornés. La question demeure donc de la possibilité de décider du caractère borné d'un réseau de Petri temporel quelconque. Nous allons esquisser ici une preuve négative, que nous espérons pédagogique.

Nous souhaitons montrer qu'une machine de Turing est simulable à l'aide des réseaux de Petri temporels. Puisqu'il n'est pas possible de décider de l'arrêt d'une machine de Turing, toute simulation d'une machine de Turing se comporte de même : il n'est donc pas possible de décider de l'arrêt de la simulation en réseaux de Petri temporels. Comme décider de l'arrêt d'une telle machine est équivalent à pouvoir décider de l'accessibilité d'un état terminal, lequel état n'est pas différent de n'importe quel état de la machine, le problème de l'accessibilité d'un état depuis l'état initial est indécidable pour les *TPN*. Ce qui à son tour empêche d'avoir une procédure de décision du caractère borné d'un réseau de Petri temporel.

Pour simuler une machine de Turing, on peut soit fournir un codage des opérations de cette machine, soit utiliser une étape intermédiaire : en codant une machine dont il a été prouvé qu'elle permet de simuler une machine de Turing. C'est le cas de la machine à deux compteurs de Minsky [Min61] qui a l'avantage de posséder un jeu d'instructions extrêmement réduit. Nous allons montrer comment coder une machine à deux compteurs grâce aux *TPN*, lequel codage est très proche de celui donné dans [JLL77].

Une machine à deux compteurs est constituée, comme son nom l'indique, de deux variables — les compteurs — à capacité infinie, ainsi que de deux opérations :

$INC(c, n)$: incrémente le compteur c et continue l'exécution avec la prochaine instruction n.

$JZDEC(c, n_1, n_2)$: teste si le compteur est à zéro, si oui va en n_1, sinon décrémente c et va en n_2.

Les figures 2.9 et 2.10 donnent un codage permettant de simuler une machine à deux compteurs à l'aide des *TPN*.

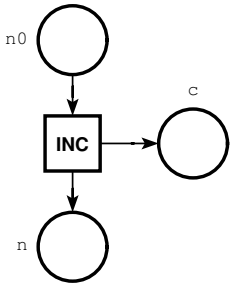


Fig. 2.9 – Codage de $INC(c, n)$

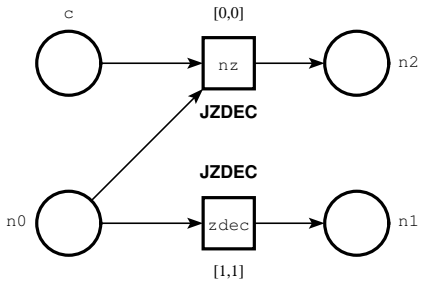


Fig. 2.10 – Codage de $JZDEC(c, n_1, n_2)$

L'instruction INC ne pose pas de difficultés particulières : n_0 est le marqueur d'instruction courante et n l'instruction suivante. Dès qu'un jeton est présent dans n_0, la transition INC est tirable et par son tir ajoute un jeton dans c ce qui pour effet d'incrémenter le compteur c et passe le marqueur d'exécution à la prochaine instruction n.

L'instruction $JZDEC$ est un peu plus complexe et doit être décomposée en deux parties : l'une partie testant si le compteur est à zéro, et l'autre décrémentant le compteur dans le cas où celui-ci possède une valeur strictement positive. La place n_0 est logiquement la précondition commune aux deux parties puisque n_0 est le marqueur d'instruction courante. Si c possède un jeton, alors les deux transitions sont sensibilisées, mais seule la transition nz est tirable, puisque sa borne maximale de tir empêche le temps d'évoluer et donc la transition *zdec* de tirer. Le tir de nz retire un jeton de la précondition c'est-à-dire de c et de n_0 pour en ajouter un dans n_2, décrémentant ainsi le compteur et déplaçant le compteur vers l'instruction suivante n_2. Dans le cas où le compteur c ne pose pas de jetons, seule la transition *zdec* est sensibilisée et devient tirable au bout d'une unité de temps. Cette valeur de progression est arbitraire : toute valeur strictement supérieure à zéro conviendrait. Le tir de cette transition ne fait que déplacer le marqueur d'exécution de n_0 vers n_1.

Les instructions d'une machine à deux compteurs sont effectuées de manière séquentielles, par conséquent ces motifs sont mis bout à bout en fusionnant les places représentant les marqueurs d'instructions à exécuter. Ainsi, la somme des jetons des places n_i est tout le temps égale à 1 et le marquage initial est le jeton repérant le marqueur de la première instruction à exécuter.

Puisque nous avons là un moyen de coder des machines à deux compteurs à l'aide des *TPN*, nous sommes sûr que la décision de la terminaison d'un modèle *TPN* est indécidable. Or détecter un état terminal n'est pas différent de détecter un état quelconque (un tel état *est* un état normal, sa seule particularité est de ne posséder aucun successeurs), ce qui implique que l'accessibilité d'un état quelconque est également indécidable. Une conséquence de cela est que l'on ne peut également pas décider de manière générale si un *TPN* est borné ou non.

Les variables représentant les compteurs doivent pouvoir prendre n'importe quelle valeur dans l'ensemble des entiers naturels, c'est à dire que le compteur, au cours de son exécution peut prendre une infinité de valeurs. Dans le contexte du *model checking*, ceci n'est pas un comportement souhaitable, puisqu'il faut un comportement représenté sous la forme d'un graphe fini (ou bien on peut utiliser une sur-approximation finie comme celle de [Fin93], ne préservant pas toutes les propriétés *LTL*). Nous nous limitons donc aux systèmes bornés. Dans ce cas-là le codage ci-dessus ne tient plus : il n'est plus possible de coder un compteur *infini*. Et un tel compteur n'est pas codable par d'autres mécanismes (comme c'est le cas si l'on étend les *TPN* à l'aide de chronomètres : [BLRV05]) car il a été prouvé que l'accessibilité est décidable pour les *TPN* bornés [BD91].

2.8 Des classes pour les *TPN*

Nous avons vu précédemment que la sémantique des réseaux de Petri est donnée par un *TTS* généralement infini. Il est par conséquent impossible de le stocker en mémoire, ce qui implique qu'il est impossible d'utiliser une technique de vérification automatique *et* exhaustive comme le *model checking* directement sur le *TTS* donnant la sémantique d'un réseau.

Le problème de représentation, lié à l'emploi de grandeurs réelles, a été résolu pour les *TPN* par [BM83] : en regroupant les états ayant un passé «similaire», il est possible d'obtenir un graphe dit de *classes*, dont il est prouvé qu'il est fini si le réseau de Petri sous-jacent est borné. Passer d'une structure infinie à une structure finie implique bien évidemment une abstraction et une abstraction est forcément plus grossière que la structure concrète. Ceci pose un problème de *préservation* de logique : la validité d'une formule logique sur l'abstraction est-elle comparable à sa validité sur la structure concrète ? La construction en classe de [BM83] assure la préservation des traces maximales. Des travaux ([YR98],[BV03],[BH06]) ont permis de raffiner cette construction, qui permettent la préservation de logiques plus puissantes, comme *CTL**, au prix d'une plus grande complexité d'analyse.

Définition 14 *Pour toute séquence tirable $\sigma \in T^*$, C_σ est l'ensemble des états définis inductivement par $C_\epsilon = \{s_0\}$ et $C_{\sigma.t} = \{s' \mid (\exists s \in C_\sigma)(\exists \theta \in \mathbb{R})(s \xrightarrow{\theta} \xrightarrow{t} s')\}$*

Les états accessibles par la même séquence de transition $\sigma.t$ sont regroupés dans la même classe $C_{\sigma.t}$ (voir figure 2.11). La préservation de la logique *LTL* tient au fait que tous les états d'une classe sont issus d'une même séquence de tir et diffèrent uniquement par l'échéancier donnant les instants de tir des transitions de la séquence. De plus, pour les *TPN*, l'écoulement du temps ne peut empêcher une transition précédemment tirable de tirer (il ne peut qu'augmenter le nombre de transitions tirables). Par conséquent les états d'une même classe sont atteignables par la même séquence de tir et sont donc indistinguables par une formule *LTL*.

2.8.1 Graphe de classe linéaire

La notion de classe définie précédemment n'est pas suffisante pour donner un graphe fini : toute boucle de transitions donne un nombre infini de séquences de transitions de longueurs arbitraires différentiables par *LTL* donnant ainsi autant de classes que de séquences (i.e. un nombre infini).

Le graphe de classe linéaire (LSCG) permet la construction d'ensembles d'états, les classes d'états, permettant d'obtenir une construction finie, donnant une abstraction du comportement du réseau de Petri préservant la logique *LTL* tels que définie précédemment. La constitution de ces ensembles d'états est possible par le calcul de la réunion des intervalles de tir de tous les états appartenant à

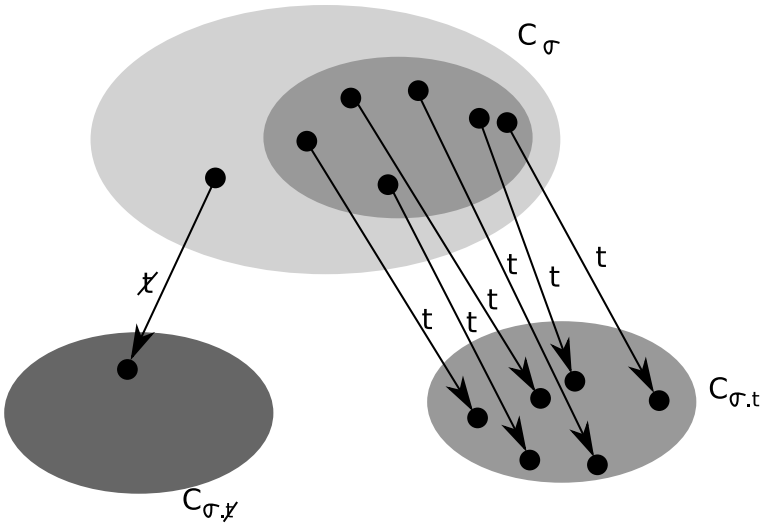


Fig. 2.11 – illustration de la définition de classe

une même classe. Une classe est donc la donnée d'un marquage et d'un système d'inéquations dont les solutions caractérisent la classe. Deux classes possédant le même marquage et le même ensemble de solutions sont équivalentes. Cette équivalence entre classes est primordiale : c'est elle qui assure que la construction est finie.

Ainsi la comparaison de classes est l'une des opérations les plus utilisée de la construction, assurant sa finitude. Afin d'obtenir une opération de comparaison dont l'efficacité est acceptable, les classes sont mises sous formes canoniques.

Algorithme 1 : Algorithme de construction du LSCG

La classe initiale $C_\epsilon = (m_0, \{\downarrow I_s(t) \leqslant \theta_t \leqslant \uparrow I_s(t) \mid \mathbf{Pre}(t) \leqslant m_0\})$;
La transition t est tirable de la classe $C_\sigma = (m, \mathcal{D}_\sigma)$ ssi :
(i) $m \geqslant \mathbf{Pre}(t)$ (t est sensibilisé par m)
(ii) Le système $\mathcal{D}_\sigma \wedge \{\underline{\theta}_t \leqslant \underline{\theta}_i \mid i \neq t \wedge m \geqslant \mathbf{Pre}(i)\}$ est consistant
Si $m_0 \xrightarrow{\sigma.t}$ alors la classe $C_{\sigma.t} = (m', \mathcal{D}_{\sigma.t})$ est calculée depuis $\mathcal{D}_\sigma$ comme suit :
1) Les contraintes de tir de t calculée en (ii) sont ajoutées à $\mathcal{D}_\sigma$
2) Pour chaque transition k sensibilisée par m' ($m \geqslant \mathbf{Pre}(k)$), une nouvelle variable $\underline{\theta}'_k$ est introduite, telle que :

$$\begin{cases} \underline{\theta}'_k = \underline{\theta}_k - \underline{\theta}_t \\ 0 \leqslant \underline{\theta}'_k \end{cases} \text{, si } k \neq t \text{ et } m - \mathbf{Pre}(t) \geqslant \mathbf{Pre}(k)$$

$\downarrow I_s(k) \leqslant \underline{\theta}'_k \leqslant \uparrow I_s(k)$ sinon
3) les variables $\underline{\theta}$ sont éliminées

L'algorithme 1 construit tout d'abord la classe initiale C_ϵ. Une classe est un couple constitué du marquage des états de la classe (ils ont tous le même marquage) et du domaine des intervalles de tir courants des transitions sensibilisées par le marquage courant. Ici, le domaine d'intervalle de tir est donné par l'ensemble des intervalles statiques de tir des transitions sensibilisées.

Ensuite, il faut tester si une transition t est tirable à partir de toute classe C_σ accessible depuis la classe initiale C_ϵ. Pour tirer, t doit bien sûr être sensibilisée (condition (i)) et il faut s'assurer que la condition $\underline{\theta}_t \leqslant \underline{\theta}_i$ ajoutée au système $\mathcal{D}_\sigma$ ne rend pas le système inconsistant, c'est-à-dire que $\mathcal{D}_\sigma$ augmenté de cette contrainte admet toujours au moins une solution. La condition $\underline{\theta}_t \leqslant \underline{\theta}_i$ signifie qu'on peut trouver une valeur $\underline{\theta}_t$ dans l'intervalle de tir de t plus petite que d'autres dans les intervalles de tir des autres transitions sensibilisées i. Il suffit alors d'attendre $\underline{\theta}_t$: puisqu'il existe des $\underline{\theta}_i$ supérieurs à $\underline{\theta}_t$ pour toute transition i sensibilisée, on est sûr de ne pas être obligé de dépasser leurs bornes maximales

de tir afin de tirer t.

Le calcul de la nouvelle classe obtenu après le tir de t est effectué en ajoutant la contrainte de tir $\underline{\theta}_t \leqslant \underline{\theta}_i$ précédente au système $\mathcal{D}_\sigma$, dont on sait maintenant qu'il est consistant puisque la transition t est tirable. A ce système $\mathcal{D}_{\sigma.t}$, on ajoute également des nouvelles variables $\underline{\theta}'_k$ pour toute transition k sensibilisée par le nouveau marquage m'. Lorsque k n'a pas été désensibilisée par le tir de t, on doit décaler son intervalle de tir par la valeur de l'attente effectuée avant le tir de t (i.e. $\underline{\theta}_t$) tout en contraignant sa valeur minimale à 0 (d'où $0 \leqslant \underline{\theta}'_k$). Lorsque k a été désensibilisée par le tir de t, son intervalle de tir est réinitialisé : $\downarrow I_s(k) \leqslant \underline{\theta}'_k \leqslant \uparrow I_s(k)$.

Finalement, on projette le système ainsi obtenu sur les variables $\underline{\theta}'_k$, ce qui revient à éliminer les variables $\underline{\theta}$ du système.

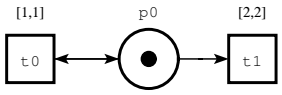
2.9 Extensions

Le modèle des *TPN* a fait l'objet de multiples extensions, dont certaines nous seront particulièrement utiles et que nous présentons maintenant.

2.9.1 read arcs

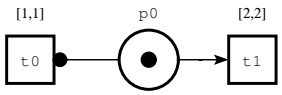
Aussi appelés *test arcs* [CH92], les *read arcs* n'ont d'intérêt que pour l'extension temporelle des réseaux de Petri. Ces arcs permettent de tester le nombre de jetons d'une place sans retirer les jetons lors du tir de la transition.

Considérons le réseau de Petri temporel (sans *read arc*) suivant :



En ne prenant en compte que son comportement atemporel (i.e. sans prendre en compte les intervalles de tir $[1,1]$ et $[2,2]$ des deux transitions), ce système peut faire une infinité de t_0, puis, éventuellement un t_1 qui termine l'exécution. Le fait de temporiser, on l'a déjà dit, réduit l'ensemble des exécutions possibles. Ici, le comportement temporel permet de tirer une infinité de fois t_0, sans jamais pouvoir tirer t_1, car à chaque tir de t_0, la transition t_1 est désensibilisée et, en conséquence, son intervalle de tir est réinitialisé à son intervalle statique.

Pour réintroduire la possibilité d'arrêter le système, nous pouvons utiliser un *read arc*. Celui-ci permet de tirer t_0 sans que celle-ci ne désensibilise t_1. Par conséquent, le comportement du réseau ci-dessous est de faire une fois t_0, puis de faire soit t_0 suivi de t_1, soit directement t_1.



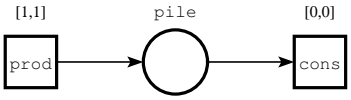
Le *read arc* est représenté par une flèche dont l'extrémité est un rond plein.

2.9.2 arcs inhibiteurs

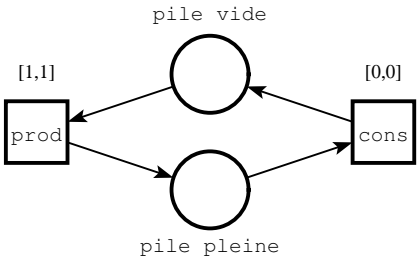
Le modèle des réseaux de Petri ne permet pas de tester si une place est vide. Mais il est souvent possible de recourir à l'utilisation de deux places *p1* et *p2* telles que $m(p1) + m(p2) = K$, où K est une constante. Ainsi, si $m(p1) = K$, alors $m(p2) = 0$, et par conséquent, en testant si la place *p1* possède K jetons, on teste du même coup si *p2* n'a pas de jetons. On dit que *p1* et *p2* sont des places *complémentaires*. Ce mécanisme impose la connaissance d'une borne K, condition qu'il n'est pas toujours possible de satisfaire.

Les arcs inhibiteurs, introduit dans [AF73], permettent d'accomplir ce test sans recourir à des places complémentaires et donc sans obligation de connaître une borne sur les jetons de la place que l'on souhaite tester à zéro. Cette extension permet aux réseaux de Petri de simuler une machine de Turing [Age75]. L'accessibilité des états est donc indécidable. Outre le fait qu'ils permettent une expressivité plus grande, ils permettent aussi d'avoir un modèle plus réduit, dans les cas où les places complémentaires sont possibles.

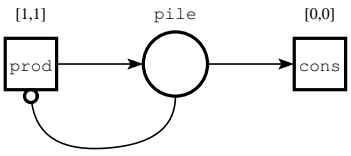
Prenons un exemple :



Ce réseau est borné, mais sa version atemporelle ne l'est pas : les temporisations l'en empêchent. Nous avons ici un modèle de producteur/consommateur simple. Le producteur empile le résultat de sa production dans la place *pile* à chaque unité de temps. Le consommateur consomme un produit dès qu'il y en a un dans la pile et cela de manière instantanée. Il est évident que la place *pile* est $1-$bornée : il est donc possible d'utiliser une place complémentaire, comme le montre la figure suivante.



Imaginons que ce réseau ne soit qu'un composant destiné à être composé avec d'autres réseaux producteurs et consommateurs, partageant la même pile de produits, mais ne possédant pas les même conditions de production ou de consommation : ceux-ci pouvant produire ou consommer plus d'un produit. La place *pile* n'est plus borné par le composant lui-même mais par le composant stockant le plus sur la pile, si une telle borne existe. Utiliser un arc inhiteur se révèle être donc très pratique pour spécifier un composant sans dépendre des bornes des composants susceptibles d'être composés avec lui.



Nous avons mis un arc inhibiteur (représenté par une flèche terminée par un rond vide) en précondition du producteur pour nous assurer que celui-ci ne produit que s'il n'y a pas de produits sur la pile, c'est-à-dire lorsque la place *pile* est vide. Comme c'est lui le seul producteur nous sommes une fois de plus assuré que la place *pile* est $1-$bornée et si ce réseau est composé avec d'autres producteurs/consommateurs, celui-ci ne produira que si la pile est vide, sans que l'on ait à s'occuper du comportement des éventuels autres composants.

2.9.3 Arcs chronomètres

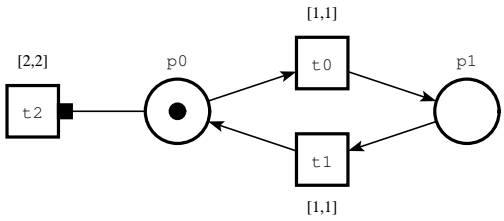
Comme nous le verrons plus en détail dans le chapitre 4, une technique permettant d'optimiser l'utilisation des ressources et du temps de réponse des tâches dans un système temps réel est d'utiliser

un ordonnancement des tâches préemptif. L'ordonnanceur peut ainsi interrompre une tâche moins prioritaire au profit d'une autre, lui permettant de reprendre son exécution lorsque sa priorité le permet.

Au niveau des réseaux de Petri, pouvoir spécifier une telle préemption implique de pouvoir suspendre l'évolution des intervalles de tir des transitions spécifiant les caractéristiques d'une tâche préemptable, mais également de pouvoir reprendre cette évolution en repartant de l'état dans lequel se trouvait l'intervalle de tir lorsque son évolution a été interrompue. C'est le principe même d'un chronomètre. Or il n'est pas possible de spécifier un chronomètre à l'aide des *TPN*. En effet, les intervalles de tir évoluent soit par un décalage vers l'origine, de manière synchrone à l'écoulement du temps, soit en prenant comme valeur celle, statique, donnée par I_s. Or pour spécifier un chronomètre, il faut pouvoir stopper le décalage de l'intervalle de tir vers l'origine *pendant* que le temps progresse, autrement dit, il faut pouvoir désynchroniser l'évolution de l'intervalle de tir d'une transition de celle du temps, et cela, alors même que la transition est sensibilisée.

Ce comportement est modélisable à l'aide des arcs chronomètres. Cette extension est une simplification des hyper-arcs inhibiteurs de [RL04]. L'utilisation des chronomètres implique l'indécidabilité de l'accessibilité des états, car il est possible de coder une machine à deux compteurs en n'utilisant que les déphasages entre horloges, lesquels déphasages sont possibles de manière suffisamment flexible avec les arcs chronomètres [BLRV05].

Une transition possède ainsi deux préconditions différentes : celle héritée des *TPN* et une nouvelle donnant un marquage partiel à respecter pour que l'intervalle de tir de la transition puisse évoluer normalement. Lorsque cette précondition n'est pas respectée, l'évolution de l'intervalle de tir est stoppée et on dit que la transition est *suspendue* ou *gelée*.



L'exemple ci-dessus présente un arc chronomètre (une flèche terminée par un carré plein), seule précondition de la transition t_2. Ainsi, le décalage de l'intervalle de tir de t_2 est interrompu lorsqu'il n'y a pas de jetons dans p_0. C'est ce qui arrive après le tir de t_0, interrompant ainsi d'une unité de temps la progression de l'intervalle de tir de t_2. Ceci a pour conséquence de ne rendre tirable t_2 qu'après trois unités de temps et non pas deux comme son intervalle statique le laisse supposer.

2.10 Conclusion

Nous avons présenté dans ce chapitre les outils fondamentaux de notre thèse : le *model checking*, une technique de vérification automatique, la logique *LTL* qui permet l'expression formelle des propriétés et le formalisme donnant le modèle pour lequel la formule doit être valide, les réseaux de Petri temporels.

Les *TPN* constituent un formalisme reconnu pour la spécification des systèmes réactifs et permettent de spécifier des systèmes temps réel complexes. Mais, comme nous allons le voir dans le chapitre suivant, certains systèmes temps réels basiques posent des problèmes de spécification et nous verrons également que la composition parallèle est une faiblesse de ce formalisme.

Nous introduirons une extension permettant d'utiliser l'information temporelle des autres transitions afin de contraindre les conditions de tir d'une transition. Nous verrons comment cette extension permet d'étendre le domaine d'application des réseaux de Petri temporels pour une modélisation des systèmes de tâches temps réel plus générale.

Nous noterons ici une extension qui peut être extrêmement utile, mais que, pour des raisons d'études théoriques, nous avons préféré ne pas employer. Il est possible d'étendre les *TPN* de manière

à pouvoir traiter des *données* de façon arbitraire, lesquelles données sont manipulables lors du tir des transitions. Cette extension «cache» la complexité de modélisation à l'aide des *TPN* sous les opérations usuelles telles que l'affectation, l'alternative, etc. Nous avons préféré garder un modèle plus "pur" en ce qui concerne la manipulation des places (avec malgré tout la concession des arcs inhibiteurs), nous permettant ainsi de mieux appréhender les forces et les faiblesses inhérentes aux *TPN*.

Une autre extension qui se révélerait très utile et néanmoins moins puissante que l'emploi de données, est celle des *flush arcs*, remettant à zéro le nombre de jetons d'une place de manière inconditionnelle. Dans le cas de cette thèse cette extension n'a pas été utilisée, toujours dans le même souci de ne garder que le strict minimum. Il est cependant à noter que lors de nos expérimentations, nous avons souvent été amené à considérer très favorablement son utilité.

Chapitre 3

Réseaux de Petri à permissions et inhibitions

Les réseaux de Petri temporels (*TPN*) permettent, nous l'avons vu au chapitre précédent, de spécifier des notions de durée indispensables pour la spécification de systèmes temps réel. Mais les *TPN* ne sont pas la pierre philosophale de la spécification de ce type de systèmes.

Nous allons voir que spécifier l'observation d'une séquence d'évènements, ainsi qu'une réaction dépendante des grandeurs temporelles observées entre ces évènements, peut ne pas être faisable de manière suffisamment fine à l'aide des *TPN*. Ce phénomène d'observation/réaction temporelle est pourtant typique des systèmes temps réel : le langage des *TPN* admet donc une limite quant à la précision avec laquelle il autorise leur spécification. Nous expliciterons dans la suite ce que signifie cette précision.

D'autre part, les systèmes temps réel admettent un caractère compositionnel natif dû à leur nature de composant contrôleur inséré dans un environnement (composant externe au système), communicant avec lui par le biais de sous-composants d'acquisitions et de réactions, dirigés par les sous-composants tâches, elles-mêmes utilisant potentiellement des sous-composants ressources. Ceci, combiné avec le fait que la modélisation des systèmes temps réel s'adapte bien à une conception incrémentale, pour laquelle un système est conçu en rajoutant (c'est-à-dire en composant) de nouvelles parties au fur et à mesure de sa spécification, implique que le formalisme utilisé devrait posséder de bonnes propriétés de compositionnalité. Même si les *TPN* admettent la possibilité d'une conception compositionnelle, celle-ci est soumise à des conditions qui nous semblent trop arbitraires pour permettre de les utiliser en pratique dans un tel contexte.

Ce chapitre présente la principale contribution théorique de cette thèse : une extension des réseaux de Petri temporels permettant de combler certaines des déficiences de ces derniers. Les deux motivations principales de cette extension sont de permettre la détermination d'un ordre entre actions autrement indéterministes car possibles aux mêmes instants et ainsi de définir une notion de priorité entre évènement simultanés. La deuxième motivation est de permettre un déplacement complet des informations temporelles portées par une transition sur d'autres transitions, qui sont ainsi en charge des contraintes appliquées sur cette première. Ce déplacement du lieu des informations permet une plus grande flexibilité de composition, qui est un aspect problématique des *TPN*.

Nous verrons également comment ce formalisme se compare face aux automates temporisés, pour qui ces problèmes de composition ne se posent pas puisque l'information temporelle est supportée par les horloges, qui sont des entités indépendantes du formalisme n'entrant pas en jeu dans la composition (la réunion des composants, effectuée par l'opération de composition, ne s'accomplit pas, même indirectement, par le biais des horloges).

Nous étudierons enfin comment la vérification de ce formalisme peut être effectuée et quels écueils sont à éviter.

3.1 Introduction - Motivation de l'extension

Le système temps réel que nous allons utiliser en guise d'exemple est couramment utilisé par une grande partie de la population : c'est le couple ordinateur-souris. Ce système est certes peu critique en ce qui concerne la sécurité des personnes ou du système lui-même, mais il possède toutes les caractéristiques d'un système temps réel : la souris capture des informations issus de l'environnement (notre mouvement de la main et l'appui sur ses boutons), pour ensuite les rediriger vers le système d'exploitation qui réagira en fonction de celles-ci. Plus précisément, nous tenterons de modéliser le fait que le système d'exploitation sait distinguer un simple clic d'un double clic selon la durée entre deux appuis sur un bouton de la souris.

Notre système doit ainsi accepter des évènements *clic* et provoquer une action *double clic* lorsque deux évènements *clic* interviennent en moins de 3 unités de temps (strict), dans le cas contraire, dès que 3 unités de temps se sont écoulées depuis le dernier évènement clic, un évènement *simple clic* est généré.

La spécification doit être la plus précise possible, c'est-à-dire que le comportement obtenu doit au minimum être faiblement temporellement bisimilaire à celui que nous venons de définir ci-dessus, ce qui assure que la plupart des logiques décidables restent utilisables pour assurer la correction du système par la validation de sa spécification.

Dans cet objectif, il ne suffit pas de préserver les séquences d'exécution (ou traces maximales, préservant la logique *LTL*), mais il faut également conserver les possibilités de branchements, qui sont indispensables afin de préserver une logique arborescente comme *CTL*.

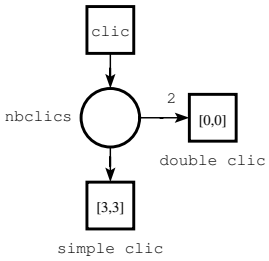


Fig. 3.1 – Première (tentative de) spécification du double clic

La première spécification (figure 3.1) correspond à très peu de choses près à ce que nous attendions. Un évènement *clic* est généré de manière arbitraire, un clic pouvant intervenir n'importe quand. Lorsqu'un *clic* a eu lieu, la transition *simple clic* commence à compter et à (exactement) 3 unités de temps génère un évènement *simple clic*. La transition *double clic* attend deux jetons dans la place *nbclics*. Dès que les deux jetons sont présents, la transition *double clic* doit être immédiatement tirée.

Qu'a donc cette spécification qui ne nous convienne pas ? Son problème est qu'elle n'est qu'une

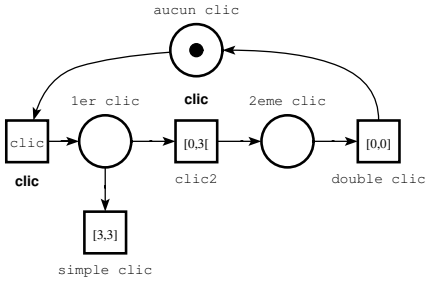


Fig. 3.2 – Deuxième (tentative de) spécification du double clic

sur-approximation du comportement attendu. Si elle permet effectivement de donner tous les comportements attendus, elle en introduit également d'autres qui sont indésirables. Observons ce qu'il se passe 3 unités de temps après l'apparition d'un évènement *clic*. La place *nbclics* possède alors un jeton et *simple clic* est tirable. *Mais ce n'est pas la seule !* En effet, *clic* est toujours tirable. Et si *clic* est tirée à 3 unités de temps, *toutes* les transitions sont alors tirables. Ce comportement n'est clairement pas celui que l'on attendait.

Nous pouvons tenter une autre approche : séparer les instances de l'évènement de *clic* en deux et temporiser la seconde de telle sorte que le deuxième clic est effectué en moins de 3 unités de temps (voir figure 3.2).

Mais là c'est clair : les transitions *clic2* et *simple clic* possèdent la même précondition, or l'intervalle de tir de *clic2* précède strictement celui de *simple clic*, en conséquence de quoi, il n'est pas possible de tirer *simple clic* ! Cette spécification est clairement incorrecte. Le lecteur pourra se demander s'il ne suffit pas d'introduire une transition servant de repère pour savoir si oui ou non on *désire* faire un double clic à l'avenir.

Dans le réseau de la figure 3.3, nous avons introduit une nouvelle place pour ne pas désensibiliser *simple clic*. Il est également nécessaire d'avoir deux transitions *pas 2 clics !* et *2 clics !* permettant de rendre le choix réversible. Un choix irréversible entraînerait que la spécification ne corresponde plus vraiment à ce que nous attendions : nous voulions initialement que les doubles clics soient possibles jusqu'à 3 unités de temps (3 exclu), or décider de manière irréversible ne correspond pas à cette sémantique : même si l'on peut retarder le choix arbitrairement, une fois le choix fait de ne pas tirer *clic* avant 3 unités de temps, il est *toujours* possible d'attendre (même si cette attente est infinitésimale) et cela sans *pouvoir* tirer *clic*, ce qui est clairement contraire à notre souhait de préserver la bisimulation. En d'autres termes, autoriser la décision de manière irréversible préserve le langage des évènements, mais ne préserve pas la bisimulation.

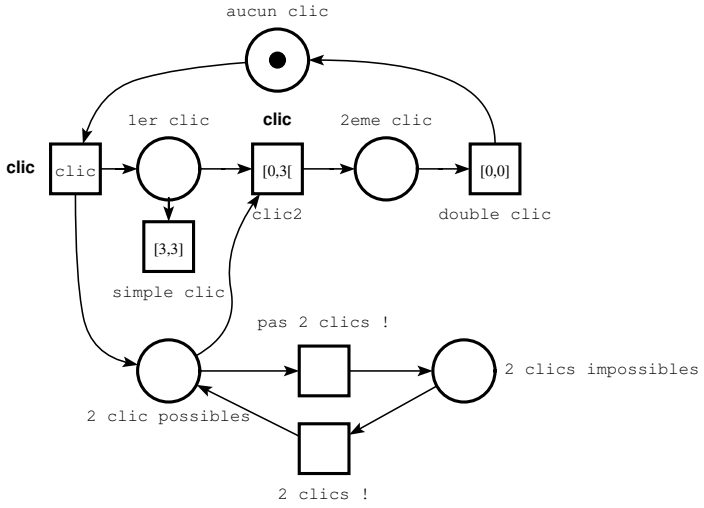


Fig. 3.3 – Troisième (tentative de) spécification du double clic

Cette spécification est plus complexe mais n'en est pourtant pas meilleure que les précédentes. En fait, nous sommes revenus à notre point de départ : il est toujours possible dans cette spécification d'avoir les deux transitions *simple clic* et *double clic* tirables en même temps car la transition *clic2* est désensibilisée par *pas 2 clics !*, ce qui remet à zéro son horloge.

Nous ne pousserons pas plus loin l'exploration, puisque celle-ci est vouée à l'échec. Nous verrons plus tard que ce type de comportement est impossible à coder avec des réseaux de Petri temporels. Pourtant, même si l'exemple du double clic peu sembler anecdotique ou prêter à sourire, ce résultat est malgré tout surprenant : c'est un système que l'on peut qualifier de réactif et surtout, plus embêtant, de temps réel. Donc il existe des systèmes temps réel qui résistent à une spécification en réseau de

Petri temporels, si l'on souhaite préserver les propriétés de branchements.

Un mot sur une autre extension souvent considérée comme apportant une notion de priorité : les arcs inhibiteurs. Leur emploi permet d'être en effet strictement plus expressif qu'avec les réseaux de Petri simples. Mais d'un point de vue du comportement *temporel*, ils ne sont pas la solution à notre problème. Car le calcul de la différence entre deux instants n'est pas stockée dans le marquage et dépend seulement des informations temporelles des transitions. Or c'est bien ce calcul qui est discriminant dans l'apparition de l'une ou l'autre des réactions du système. En d'autres termes, en présence d'arcs inhibiteurs, le théorème 1 (voir page 44), stipulant que le passage du temps ne peut empêcher le tir d'une transition déjà tirable, reste toujours vrai, alors que nous souhaitons clairement qu'il soit faux dans notre exemple.

Quel est la véritable source du problème de notre exemple ? Il vient du fait que nous ne pouvons empêcher un évènement d'intervenir *logiquement* avant un autre, tout en étant *temporellement* simultané, sans le recours au marquage. Dans le cas contraire, il suffirait, pour résoudre notre problème, de modifier le premier exemple en empêchant *clic* de tirer *logiquement* avant *simple clic*, mais sans l'empêcher de tirer au même instant temporel. Autrement dit, il suffirait de forcer l'ordre d'exécution à l'instant 3 en forçant le tir de *simple clic* avant celui de *clic*, et cela sans avoir modifié le marquage.

3.2 Réseaux de Petri temporels à permissions et inhibitions

Les réseaux de Petri temporels à permissions et inhibitions (*ipTPN*) étendent les *TPN* à l'aide de deux nouvelles relations entre les transitions. Afin de comparer ce nouveau formalisme avec ses prédécesseurs, nous utilisons un alphabet d'actions et une fonction d'étiquetage des transitions. $\mathbf{I}^+$ est l'ensemble des intervalles à bornes rationnelles non-négatives. Si $i \in \mathbf{I}^+$, $\downarrow i$ donne sa borne inférieure, et $\uparrow i$ sa borne supérieure si i est bornée, ou ∞ dans le cas contraire. Pour tout $\theta \in \mathbf{R}^+$, $i \dot{-} \theta$ est l'intervalle $\{x - \theta \mid x \in i \wedge x \geqslant \theta\}$.

Définition 15 *Un* réseau de Petri temporel à permissions et inhibitions *(*ipTPN *en abrégé) est un n-uplet* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s, F, A, \Sigma^\epsilon, \mathcal{L}\rangle$ *dans lequel :*

- $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s, \Sigma^\epsilon, \mathcal{L}\rangle$ *est un réseau de Petri temporel,*
- $F \subseteq T \times T$ *est la relation d'inhibition (F pour* forbid *en anglais),*
- $A \subseteq T \times T$ *est la relation de permission (A pour* allow *en anglais),*
- Σ *est un ensemble fini d'*actions *(ou* étiquettes*), qui ne contient pas l'action* silencieuse ϵ*,*
- $L : T \to \Sigma^\epsilon$ *est une fonction appelée fonction* d'étiquetage*.*

$(t_1, t_2) \in F$ est noté $t_1 \multimap t_2$ (t_1 inhibe (ou interdit le tir de) t_2) et $(t_1, t_2) \in A$ est noté $t_2 \bullet\!\!-\!\!\circ\, t_1$ (t_1 autorise (ou permet) le tir de t_2). Une transition sera dite *T-sensibilisée* si zéro appartient à son intervalle de tir. Dire d'une transition qu'elle est tirable est équivalent à dire qu'elle est T-sensibilisée dans le formalisme des *TPN* (mais pas dans celui des *ipTPN* !).

Définition 16 *La sémantique de l'*ipTPN $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s, F, A, \Sigma^\epsilon, \mathcal{L}\rangle$ *est le système de transitions temporisé* $\langle S, (m_0, I_s^0), \Sigma, \rightarrow\rangle$ *où :*

- I_s^0 *est la fonction* I_s *restreinte aux transitions sensibilisées par* m_0
- *les états de* S *sont les paires* (m, I) *dans lesquelles* m *est un marquage et* $I : T \to \mathbf{I}^+$ *associe un intervalle temporel à chaque transition sensibilisée par* m*,*
- $(m, I) \xrightarrow{\mathcal{L}(t)} (m', I')$ *ssi* $t \in T$ *et*
  1) $\mathbf{Pre}(t) \leqslant m$
  2) $0 \in I(t)$
  3) $(\forall k \in T)(\mathbf{Pre}(k) \leqslant m \wedge 0 \in I(k) \Rightarrow \neg(k \multimap t))$
  4) $(\forall k \in T)(\mathbf{Pre}(k) \leqslant m \wedge 0 \notin I(k) \Rightarrow \neg(k \bullet\!\!-\!\!\circ\, t))$
  5) $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$
  6) $(\forall k \in T \setminus \{t\})(\mathbf{Pre}(k) \leqslant m - \mathbf{Pre}(t) \Rightarrow I'(k) = I(k))$
  7) $(\forall k \in T)(\mathbf{Pre}(k) \leqslant m' \Rightarrow I'(k) = I_s(k))$

– $(m, I) \xrightarrow{d} (m, I')$ *ssi*
8) $(\forall k \in \mathcal{E}n(m))(d \leqslant \uparrow I(k))$
9) $(\forall k \in \mathcal{E}n(m))(I'(k) = I(k) \dot{-} d)$

Une transition t peut tirer à partir de l'état (m, I) si t est sensibilisée par m, si elle est tirable instantanément et si aucune autre transition inhibant t ne satisfait ces conditions. Dans l'état destination, les transitions qui restent sensibilisées pendant le tir de t (t exclue) gardent leur intervalle. Les autres transitions (dites nouvellement sensibilisées) sont associées à leur intervalle de tir statique.

Une transition continue valant d unités de temps est possible ssi d ne dépasse pas les bornes maximales $\uparrow I(k)$ des transitions k sensibilisées (autrement dit, appartenant à $\mathcal{E}n(m)$).

Les conditions 3) et 4) sont les nouvelles conditions de tir introduites par l'ajout des relations F et A. La condition 3) stipule que toute transition sensibilisée par le marquage courant m ($m \geqslant \mathbf{Pre}(k)$) qui est T-sensibilisée ($0 \in I(k)$) ne contraint pas t par la relation d'inhibition $\neg(k \multimap t)$. On peut le voir dans l'autre sens : t n'est tirable que si toute transition k sensibilisée par le marquage m et inhibant t ($k \multimap t$) n'est pas T-sensibilisée.

La condition 4) exprime la condition inverse : t est tirable s'il n'existe aucune transition k sensibilisée par le marquage courant m et T-désensibilisée telle que $k \bullet\!\!-\!\!\circ t$. Autrement dit, si $k \bullet\!\!-\!\!\circ t$ alors t n'est tirable que si k est T-sensibilisée.

3.3 Expressivité

Dans ce qui suit, le lecteur remarquera peut-être que nous privilégions la relation d'inhibition sur celle de permission. Ceci est une conséquence chronologique : c'est elle qui a été introduite en premier sous le nom de relation de priorités. Nous allons reprendre dans la suite les principaux résultats de [BPV06], [BPV07] et [PBV09].

3.3.1 Introduction

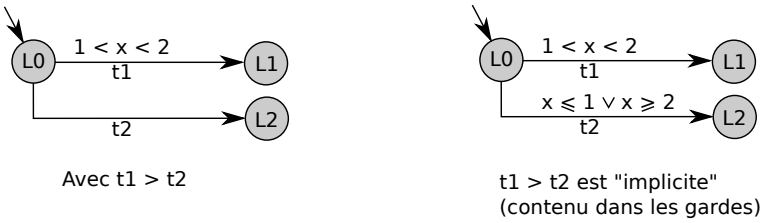


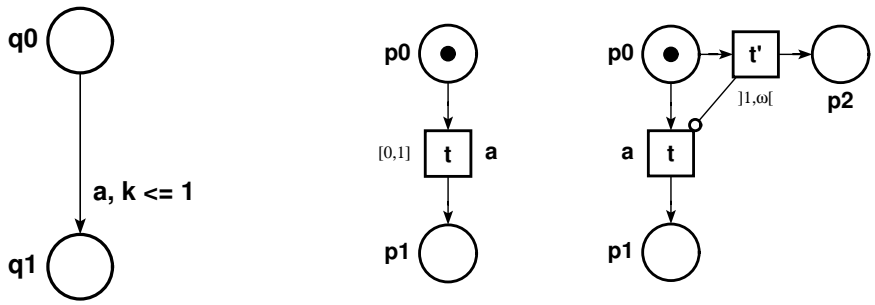
Fig. 3.4 – La prise en compte des priorités par les TA est native

Les priorités (équivalents à la relation d'inhibition) pour les automates temporisés (abrégé en TA, voir la section 3.3.4 pour une définition détaillée de ce formalisme) sont codables très simplement : pour qu'une transition soit tirée en priorité, il suffit de modifier les gardes des transitions moins prioritaires de façon à ce que celles-ci soient empêchées de tirer aux instants où la transition plus prioritaire est tirable. Un exemple de ce codage est donné en figure 3.4. Si ce codage n'a rien de compliqué, sa taille peut être significativement plus importante que la version avec priorités explicites, selon la formule de la garde de la transition plus prioritaire.

Contrairement aux automates temporisés, la relation d'inhibition ajoute de l'expressivité aux *TPN* bornés. Pour illustrer cela, considérons l'automate et le réseau de la figure 3.5.

Comme mentionné dans [BCH$^+$05b], il n'existe aucun *TPN* qui soit temporellement bisimilaire, même faiblement, à l'automate 3.5(a). En particulier, le réseau 3.5(b) ne l'est pas : quand le temps peut progresser d'une quantité arbitraire à la location q_0, il ne peut dépasser une unité de temps dans le *TPN* 3.5(b).

L'argument essentiel de l'impossibilité de spécifier finement l'exemple du double clic de l'introduction ou de traduire l'automate 3.5(a) en *TPN*, vient de ce que les *TPN* possèdent la caractéristique suivante (tiré de [BCH$^+$05b]) :



(a) (b) (c)

Fig. 3.5 – Expressivité de la relation d'inhibition dans les *TPN*

Théorème 1 Le passage du temps ne peut empêcher le tir d'une transition déjà tirable (WCDT)[1]

Si dans l'état (m, I), la transition t est tirable : $(m, I) \xrightarrow{t}$ et s'il est possible d'attendre $d \geqslant 0$ unités de temps, pour arriver dans l'état (m, I') : $(m, I) \xrightarrow{d} (m, I')$, alors la transition t est toujours tirable à partir de l'état (m, I').

Preuve La transition $\xrightarrow{d}$ ne change pas le marquage et d doit nécessairement être inférieur à la borne maximale de toutes les transitions sensibilisés, et puisque $(m, I) \xrightarrow{t}$, alors $m \geqslant \mathbf{Pre}(t)$, par conséquent la transition t est forcément toujours tirable à (m, I'). □

Pour empêcher le tir d'une transition tirable dans les *TPN*, il faut la désensibiliser. C'est pour cela qu'il nous a été impossible d'exhiber une spécification du double clic acceptable. Revenons un instant sur cet exemple. Nous souhaitions spécifier que 3 unités de temps après un évènement *clic*, si un autre *clic* intervient, alors il ne faut pas le considérer comme un *double clic*, c'est-à-dire qu'il ne faut pas que *double clic* soit tirable.

Nous ne pouvons appliquer directement le théorème *WCDT* à l'exemple du double clic, mais il va nous être utile. Nous utiliserons plutôt le théorème suivant :

Théorème 2 *Si il existe une séquence de transitions $t_1 \dots t_n$, et une séquence de délais $\theta_1 \dots \theta_n$ telles que $(m, I) \xrightarrow{t_1\theta_1 \dots t_n\theta_n} (m', I')$ et de l'état (m', I') on peut tirer la transition k et si, de plus, il existe également une séquence de délais $\theta'_1 \dots \theta'_n$ telle que $(\forall i \leqslant n)(\theta'_i \geqslant \theta_i)$ et $(m, I) \xrightarrow{t_1\theta'_1 \dots t_n\theta'_n} (m'', I'')$, alors k est tirable à partir de (m'', I'').*

Preuve

Comme la séquence de transitions menant à (m', I') et à (m'', I'') est la même, alors $m' = m''$. Par conséquent, les deux états ne diffèrent que par I' et I''. Puisque $(m', I') \xrightarrow{k}$, alors $0 \in I'(k)$. Comme $(\forall i \in [1, n])(\theta_i \leqslant \theta'_i)$, alors les domaines de tir de I'' sont forcément plus décalés vers l'origine que ceux de I' : $(\forall t)(\exists x \in I''(t))(\forall y \in I'(t))(x \leqslant y)$. Par conséquent $0 \in I''(k)$ et donc $(m'', I'') \xrightarrow{k}$. □

Comme *Double clic* est tirable par une séquence *clic*.θ.*clic*, avec $\theta < 3$, et que la spécification impose qu'il doit être possible de faire la séquence *clic*.3.*clic*, alors *Double clic* doit nécessairement être tirable également dans cette séquence. Nous voyons donc que nous sommes dans une impossibilité intrinsèque aux *TPN*.

Revenons à l'automate de la figure 3.5. Nous pouvons le coder à l'aide de la relation d'inhibition de notre nouvelle extension. Cette relation est représentée par un arc entre transitions terminé par un rond creux (sa représentation graphique est semblable à celle des arcs inhibiteurs). Sur l'*ipTPN* 3.5(c), nous avons $t' \multimap t$. La transition t' est silencieuse (son étiquette est ϵ) et tirable strictement après 1 unité de temps, la transition t est étiquetée par a et son intervalle de tir est le plus lâche

[1] Waiting Cannot Disable Transitions

possible ($[0, \infty[$), cela signifie que l'on n'impose directement aucune condition temporelle particulière sur t. Mais son espace temporel de tir n'en reste pas moins contraint : elle peut tirer à n'importe quel moment compris entre 0 et 1 unité de temps (inclus), mais pas plus tard, puisque t' devient alors T-sensibilisée (et, ici, tirable) et inhibe donc t. Le réseau 3.5(c) est donc faiblement temporellement bisimilaire à l'automate 3.5(a). Nous verrons par la suite qu'il est possible de faire mieux.

Du moment qu'il n'y pas de transitions t telles que $t \multimap t$, la relation d'inhibition ne peut empêcher le temps de progresser. Dans tous les cas, elle enrichie les contraintes de tir des transitions.

3.3.2 Composition des *ipTPN*

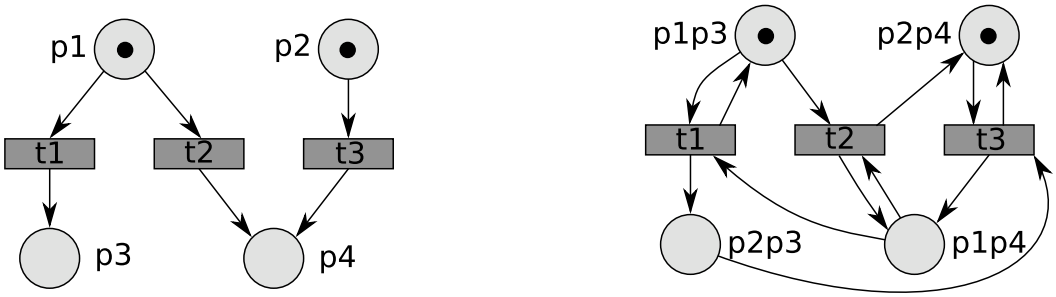
La composition de réseaux est très utile et permet la conception de systèmes de manière incrémentale. Nous allons décrire ici deux opérateurs de fusion, l'un de places, l'autre de transitions, qui nous permettront de définir une opération de composition que nous utiliserons partiellement dans un premier temps afin d'étudier l'expressivité des *ipTPN* en comparant la composition parallèle (par transitions) des *TPN* et des *ipTPN*. Dans un deuxième temps, cette composition nous servira à assembler les blocs sémantique du langage dédié aux systèmes de tâches temps réel proposé dans cette thèse dans le chapitre 5.

Définition 17 *Soit* $N = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0 \rangle$ *un réseau de Petri, et deux ensembles* $A \subseteq P$ *et* $B \subseteq P$ *tel que* $A \cap B = \emptyset$.

La fusion des places de A *et* B *de* N *donne le réseau* $N' = \langle P', T', \mathbf{Pre}', \mathbf{Post}', m_0' \rangle$, *avec :*

- $P' = (P \setminus (A \cup B)) \cup (A \times B)$[1]
- $(\forall p \in P \setminus (A \cup B))(\forall t \in T)(\mathbf{Pre}'(t)(p) = \mathbf{Pre}(t)(p) \wedge \mathbf{Post}'(t)(p) = \mathbf{Post}(t)(p))$
- $(\forall p = (a, b) \in A \times B)(\forall t \in T)(\mathbf{Pre}'(t)(p) = \mathbf{Pre}(t)(a) + \mathbf{Pre}(t)(b) \wedge \mathbf{Post}'(t)(p) = \mathbf{Post}(t)(a) + \mathbf{Post}(t)(b))$
- $(\forall p \in P \setminus (A \cup B))(m_0'(p) = m_0(p))$
- $(\forall p = (a, b) \in A \times B)(m_0'(p) = m_0(a) + m_0(b))$

L'opération est notée $\oplus_B^A(N) = N'$



(a) Un réseau de Petri N

(b) Le réseau $\oplus_{\{p3,p4\}}^{\{p1,p2\}}(N)$

Fig. 3.6 – Exemple de transformation par la fusion de place $\oplus$

Intuitivement, l'opération $\oplus_{\{p3,p4\}}^{\{p1,p2\}}(N)$, dont le résultat est donné par la figure 3.6 (b), fusionne les places de $\{p1, p2\}$ avec les places de $\{p3, p4\}$ par le produit cartésien de ces ensembles. Ainsi, les places $p1$ et $p2$ ne sont pas destinées à être fusionnées entre elles, mais avec l'une et puis l'autre des places $p3$ et $p4$.

Il est plus facile de comprendre cette opération en considérant que les places des deux ensembles A et B sont deux interfaces de «composants» à inter-connecter. Bien qu'il ne soit pas question ici de plusieurs composants, nous verrons plus tard qu'il suffit d'une opération de réunion de réseaux

[1] par soucis de concision, nous confondrons l'ensemble $A \times B$ avec la fonction associant un symbole (dans T) à chacun de ses éléments de $A \times B$

disjoints pour que $\oplus$ devienne applicable en tant que deuxième étape lors de la définition de l'opération de composition de réseaux.

Même si l'opération permet bien plus, le cas typique d'application d'une telle opération est donné par la figure 3.7, où l'on peut distinguer deux «composants» (en admettant qu'ils ont été réunis au sein du même réseau par une opération de réunion de réseaux), que l'on enchaîne à l'aide de la fusion $\oplus^{\{p2\}}_{\{p3\}}(R)$, en fusionnant les places $p2$ et $p3$.

(a) Réseau R (b) Réseau $\oplus^{\{p2\}}_{\{p3\}}(R)$

Fig. 3.7 – Cas d'utilisation typique de $\oplus$

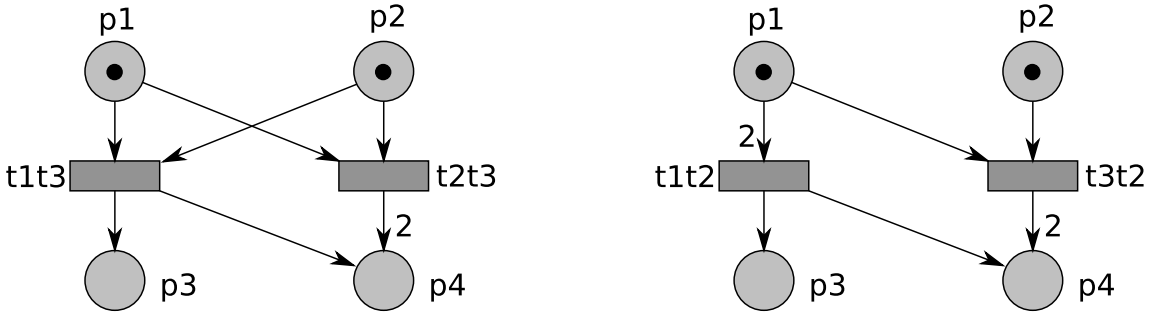
Il est à remarquer que la fusion par place ne prend en compte que les informations ayant trait aux places : marquages, préconditions, postconditions. Ceci implique que toute extension ne modifiant pas ces structures n'est pas influencée (du moins directement) par la composition par place. C'est le cas des *TPN* et des *ipTPN*. Il n'est donc pas nécessaire de la redéfinir pour ces deux formalismes.

Définition 18 *Soit* $N = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0\rangle$ *un réseau de Petri, et deux ensembles* $A \subseteq T$ *et* $B \subseteq T$ *tel que* $A \cap B = \emptyset$.

La fusion des transitions de A *et* B *de* N *donne le réseau* $N' = \langle P', T', \mathbf{Pre}', \mathbf{Post}', m_0\rangle$, *avec :*

- $T' = T \setminus (A \cup B) \cup (A \times B)$
- $(\forall t \in T \setminus (A \cup B))(\forall p \in P)(\mathbf{Pre}'(t)(p) = \mathbf{Pre}(t)(p) \wedge \mathbf{Post}'(t)(p) = \mathbf{Post}(t)(p))$
- $(\forall t = (a,b) \in A \times B)(\forall p \in P)(\mathbf{Pre}'(t)(p) = \mathbf{Pre}(a)(p) + \mathbf{Pre}(b)(p) \wedge \mathbf{Post}'(t)(p) = \mathbf{Post}(a)(p) + \mathbf{Post}(b)(p))$

L'opération est notée $pn \otimes^A_B (N) = N'$



(a) Le réseau $pn \otimes^{\{t1,t2\}}_{\{t3\}} (N)$ (b) Le réseau $pn \otimes^{\{t1,t3\}}_{\{t2\}} (N)$

Fig. 3.8 – Exemples de transformation par la fusion de transitions $pn\otimes$

L'opération de fusion de transitions $pn\otimes$ est très similaire à $\oplus$. La fusion de transitions s'opère également d'après le produit cartésien des ensembles A et B de l'opération $pn \otimes^A_B (x)$. Et de la même façon, cette opération doit être comprise comme une partie de l'opération de composition que nous définirons plus tard, permettant de fusionner les transitions à l'interface de deux composants qui auront été au préalable réunis en un seul réseau.

A l'inverse de l'opération $\oplus$, l'opération $pn\otimes$ est sous-spécifiée pour les réseaux de Petri temporels, car elle ne spécifie pas quel doit être l'intervalle de tir de la fusion de deux transitions portant un intervalle de tir non trivial.

Définition 19 *Soit $N = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s\rangle$ un réseau de Petri temporel, et deux ensembles $A \subseteq T$ et $B \subseteq T$ tel que $A \cap B = \emptyset$.*

La fusion des transitions de A et B de N donne le réseau de Petri temporel suivant : $N' = \langle P, T', \mathbf{Pre}', \mathbf{Post}', m_0, I'_s\rangle$ avec

- *T', $\mathbf{Pre}'$ et $\mathbf{Post}'$ suivent la règle de composition $pn \otimes_B^A (\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0\rangle)$*
- *$(\forall t \in T \setminus (A \cup B))(\forall p \in P)(I'_s(t) = I_s(t))$*
- *$(\forall t = (a, b) \in A \times B)(\forall p \in P)(I'_s(t) = I'_s(a) \cap I'_s(b))$*

L'opération est notée $tpn \otimes_B^A (N) = N'$

Cette définition ne pose pas de difficulté majeure. Elle n'est pas définie dans le cas où $I'_s(a) \cap I'_s(b) = \emptyset$. Une illustration de cette intersection d'intervalle lors de la fusion par $tpn\otimes$ est donnée par la figure 3.9 où l'opération plus générale $\otimes$ est utilisée.

En ajoutant les deux nouvelles relations F et A aux TPN, la fusion de transitions s'en trouve bien évidemment affectée :

Définition 20 *Soit $N = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s, F, A\rangle$ un réseau de Petri temporel à inhibitions/permissions, et deux ensembles $A \subseteq T$ et $B \subseteq T$ tel que $A \cap B = \emptyset$.*

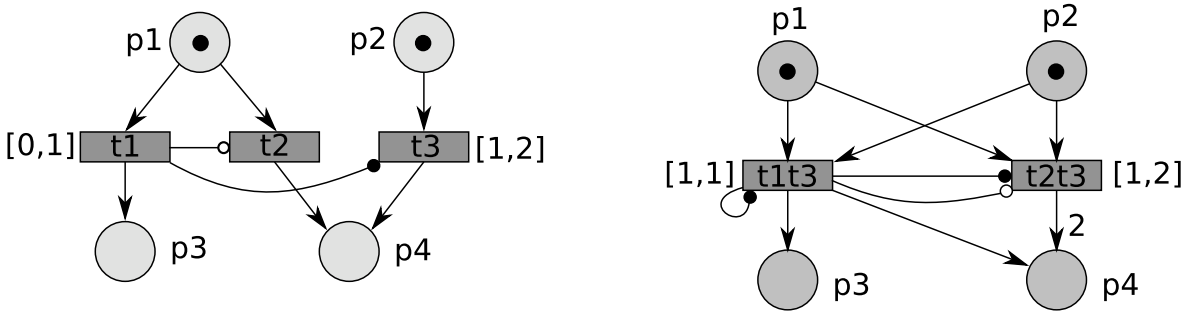
Soit $\mathcal{B} : T \to \mathcal{P}(T')$, la fonction associant à une transition de T un ensemble de transition de T', de telle sorte que :

- *$(\forall t \in T \setminus (A \cup B))(\mathcal{B}(t) = \{t\})$*
- *$(\forall t \in A \cup B)(\mathcal{B}(t) = \{(a, b) \mid (a, b) \in A \times B \wedge (a = t \vee b = t)\}$*

La fusion des transitions de A et B de N donne le réseau de Petri temporel suivant : $N' = \langle P, T', \mathbf{Pre}', \mathbf{Post}', m_0, I'_s, F', A'\rangle$ avec

- *T', $\mathbf{Pre}'$ et $\mathbf{Post}'$ suivent la règle de composition $pn \otimes_B^A (\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0\rangle)$*
- *I'_s suit la règle de composition $tpn \otimes_B^A (\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I'_s\rangle)$*
- *$(\forall f \in \{F, A\})((t_1, t_2) \in f \Rightarrow (\forall u, v \in \mathcal{B}(t_1) \times \mathcal{B}(t_2))((u, v) \in f'))$*

L'opération est notée $\otimes_B^A(N) = N'$



(a) Le réseau N'

(b) Le réseau $\otimes_{\{t3\}}^{\{t1,t2\}}(N')$

Fig. 3.9 – Exemple de transformation par la fusion de transitions $\otimes$

La fonction $\mathcal{B}(t)$ associe à la transition t (appartenant au réseau avant transformation) l'ensemble des transitions fusionnées pour lesquelles t a participé. Ainsi, pour le réseau N' de la figure 3.9 (a), on obtient :

- $\mathcal{B}(t1) = \{(t1, t3)\}$
- $\mathcal{B}(t2) = \{(t2, t3)\}$
- $\mathcal{B}(t3) = \{(t1, t3), (t2, t3)\}$

Par ailleurs, on a $(t1,t2) \in F$ et $(t1,t3) \in A$ et comme $\mathcal{B}(t1)$ (resp. $\mathcal{B}(t2)$) est l'ensemble des transitions fusionnées pour lesquelles $t1$ (resp. $t2$) a participé, alors les transitions de $\mathcal{B}(t1)$ inhibent les transitions de $\mathcal{B}(t2)$. D'où $\mathcal{B}(t1) \times \mathcal{B}(t2) \in F'$, c'est-à-dire que $(t1,t3) \multimap (t2,t3)$. Et le même raisonnement est utilisé pour la relation A' : $\mathcal{B}(t1) \times \mathcal{B}(t3) \in A'$, c'est-à-dire $(t1,t3) \bullet\!\!-\!\!\circ (t1,t3)$[1] et $(t1,t3) \bullet\!\!-\!\!\circ (t2,t3)$.

En vue de la définition de la composition des réseaux *ipTPN*, nous allons nous servir d'une opération de réunion de réseau, qui, comme nous l'avons déjà dit permettra d'utiliser les opérations de fusion que nous venons de définir.

Définition 21 réunion de réseaux

Soient deux ipTPN

$N_a = \langle P^a, T^a, \mathbf{Pre}^a, \mathbf{Post}^a, m_0^a, I_s^a, F^a, A^a, \Sigma_a^\epsilon, \mathcal{L}^a \rangle$ *et*

$N_b = \langle P^b, T^b, \mathbf{Pre}^b, \mathbf{Post}^b, m_0^b, I_s^b, F^b, A^b, \Sigma_b^\epsilon, \mathcal{L}^b \rangle$

avec $P^a \cap P^b = \emptyset$ *et* $T^a \cap T^b = \emptyset$.

La réunion des deux réseaux N_a *et* N_b *est le réseau* $N_a \circ N_b = \langle P^a \cup P^b, T^a \cup T^b, \mathbf{Pre}^a \cup \mathbf{Pre}^b, \mathbf{Post}^a \cup \mathbf{Post}^b, m_0^a \cup m_0^b, I_s^a \cup I_s^b, F^a \cup F^b, A^a \cup A^b, \Sigma_a \cup \Sigma_b \cup \{\epsilon\}, \mathcal{L}^a \cup \mathcal{L}^b \rangle$.

L'opération de composition de réseaux va utiliser l'étiquetage des réseaux qui est donné par une fonction d'étiquetage applicable aussi bien sur les places que sur les transitions du réseau (utiliser un même label sur une place et une transition ne signifie rien de spécial). Les places et les transitions qui portent la même étiquette dans des réseaux différents sont combinées comme décrit dans la définition 23 ci-dessous.

Les fonctions d'étiquetages peuvent retourner la valeur neutre ϵ signifiant par là que l'élément ne porte pas d'étiquette. Nous nous servirons donc d'une égalité modifiée pour tenir compte de cela, en ne considérant que les étiquettes différentes de ϵ. Ainsi, deux étiquettes ne seront considérées comme ϵ-égales que si elles sont effectivement égales mais également différentes de ϵ.

Définition 22 ϵ-égalité

Soit Σ *un alphabet quelconque contenant* ϵ *et* $a, b \in \Sigma$. $a =^\epsilon b$ *ssi* $a = b \neq \epsilon$.

Définition 23 composition de réseaux

Soient deux réseaux de Petri temporels à inhibitions/permissions étiquetés

$N_a = \langle P^a, T^a, \mathbf{Pre}^a, \mathbf{Post}^a, m_0^a, I_s^a, F^a, A^a, \Sigma_a^\epsilon, \mathcal{L}^a \rangle$ *et*

$N_b = \langle P^b, T^b, \mathbf{Pre}^b, \mathbf{Post}^b, m_0^b, I_s^b, F^b, A^b, \Sigma_b^\epsilon, \mathcal{L}^b \rangle$.

Soient $\Delta_Y^X = \{S \subseteq X | (\forall (x,y) \in S^2)(\mathcal{L}^a(x) =^\epsilon \mathcal{L}^a(y) \wedge (\exists y \in Y)(\mathcal{L}^a(x) =^\epsilon \mathcal{L}^b(y))\}$
et $\nabla_Y^X = \{y \in Y | (\forall x \in X)(\mathcal{L}^a(x) =^\epsilon \mathcal{L}^b(y))\}$

La composition des deux réseaux étiquetés est donnée par :

$$N_a \odot N_b = \bigotimes_{x \in \Delta_{T^b}^{T^a}}^{\nabla_{T^b}^x} \bigoplus_{x \in \Delta_{P^b}^{P^a}}^{\nabla_{P^b}^x} (N_a \circ N_b) \tag{3.1}$$

La définition est dense et requiert quelques commentaires.

Tout d'abord, il convient de préciser que

$$\bigotimes_{x \in A}^{f(x)} N = \otimes_{x_1}^{f(x_1)} ... \otimes_{x_n}^{f(x_n)} (N) \text{ avec } A = \bigcup_{i=1}^{n} \{x_i\} \tag{3.2}$$

$$\bigoplus_{x \in A}^{f(x)} N = \oplus_{x_1}^{f(x_1)} ... \oplus_{x_n}^{f(x_n)} (N) \text{ avec } A = \bigcup_{i=1}^{n} \{x_i\} \tag{3.3}$$

[1] une boucle $t \bullet\!\!-\!\!\circ t$ n'apporte que des informations redondantes et peut être supprimée

Δ_Y^X donne la partition de l'ensemble X selon l'équivalence $=^\epsilon$, sous la condition que les éléments de ces partitions possèdent au moins un élément équivalent dans Y. Par Δ_Y^X on récupère ainsi les ensembles de transitions qui vont servir à la fusion des transitions de X avec les transitions de Y. On peut aussi dire que chaque partie de Δ_Y^X est une classe d'équivalence pour $=^\epsilon$ dans X, conditionnée par l'existence d'éléments équivalents dans Y.

∇_Y^X définie l'ensemble des éléments de Y qui sont équivalents à des éléments de X. L'ensemble $\nabla_{T^b}^x$ donne ainsi les éléments de T^b qui portent une étiquette commune avec des éléments de x.

La composition s'opère alors par la fusion de places — puis de transitions — équivalentes, de manière incrémentale, en partant de la réunion des deux réseaux.

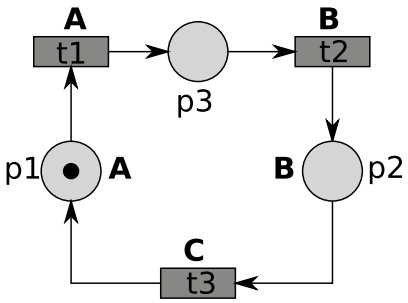


Fig. 3.10 – Réseau R

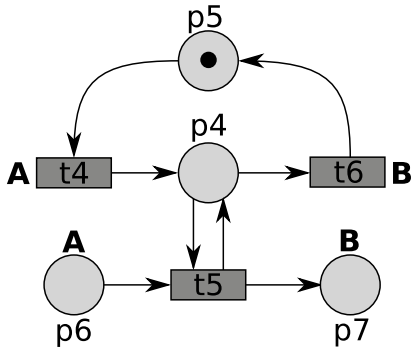


Fig. 3.11 – Réseau R'

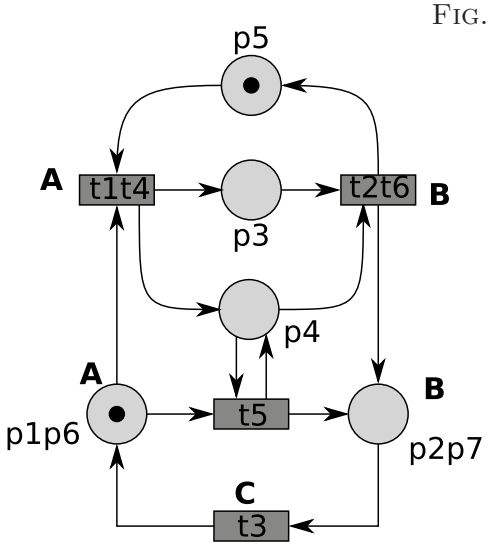


Fig. 3.12 – Réseau $R \odot R'$

La première étape du calcul de la composition $R \odot R'$ est de déterminer $\Delta_{T^b}^{T^a}$: c'est la partition de l'ensemble T^a telle que les éléments de chaque partie portent la même étiquette (sont ϵ-égaux), en enlevant les parties qui n'ont pas d'équivalents dans T^b. La partition de T^a selon $=^\epsilon$ est $\{\{t_1\},\{t_2\},\{t_3\}\}$, mais la partie $\{t_3\}$, dont le seul élément t_3 est étiqueté par **C** n'a pas d'équivalent dans T^b. D'où $\Delta_{T^b}^{T^a} = \{\{t_1\},\{t_2\}\}$.

Pour chaque partie de $\Delta_{T^b}^{T^a}$, il faut déterminer l'ensemble des éléments (de même nature : ici des transitions) équivalents : c'est ce qui est donné par $\nabla_{T^b}^{\{t_1\}} = \{t_4\}$ et $\nabla_{T^b}^{\{t_2\}} = \{t_6\}$. On fait de même pour les places : $\Delta_{P^b}^{P^a} = \{\{p_1\},\{p_2\}\}$, $\nabla_{P^b}^{\{p1\}} = \{p_6\}$ et $\nabla_{P^b}^{\{p_2\}} = \{p_7\}$. L'opération $R \odot R'$ peut être alors complètement développée :

$$R \odot R' = \otimes_{\{t_1\}}^{\{t_4\}} \otimes_{\{t_2\}}^{\{t_6\}} \oplus_{\{p_1\}}^{\{p_6\}} \oplus_{\{p_2\}}^{\{p_7\}} (R \circ R') \tag{3.4}$$

3.3.3 Les *ipTPN* pour rendre les *TPN* composables

Nous avons tous tendance, face à un problème complexe, à vouloir le décomposer en sous-parties plus simples. Non que la complexité du système s'en trouve réduite, mais il est ainsi plus facile d'appréhender les aspects de ce système sous des angles plus accessibles. C'est la base des méthodes de conception incrémentales, basées sur l'ajout successif des sous-systèmes ainsi produits. La méthodologie que nous utilisons dans cette thèse est particulièrement sujette à de telles décompositions.

Le modèle des réseaux de Petri admet intrinsèquement un fonctionnement parallèle (ou concurrent) et peut ultimement être décomposé jusqu'à n'avoir plus qu'une seule transition (et toutes les places nécessaires à ses pré/post-conditions) par composant. Cette décomposition n'est pas toujours la plus adéquate en terme de conception, mais sa faisabilité ne pose aucun problème théorique. L'une des meilleures démonstrations de ce fait est le langage algébrique à base de *boîtes* Petri, le *Petri Box Calculus* [BDH92], où l'élément de base est une transition avec une place en entrée et en sortie, qui va être manipulé par des opérateurs de composition (proches de ceux que nous venons de définir) tels que l'on puisse construire plus ou moins aisément tout type de réseaux.

S'il est reconnu que les réseaux de Petri sont aisément composables, il est par contre également connu que rajouter une information temporelle sur les transitions restreint fortement les possibilités de composition très larges qui existaient dans le cadre atemporel. En effet, comment composer des transitions portant un intervalle temporel non trivial ? Comment concilier des intervalles différents ? La méthode la plus courante, nous venons de le voir, est de prendre l'intersection des intervalles temporels des transitions composées. Mais ceci n'est qu'un pis-aller posant plus de problèmes qu'il n'en résout. Cela signifie que l'on ne peut pas toujours effectuer la composition : si les intervalles sont disjoints le résultat serait incohérent. Mais, plus grave, cette composition ne présente pas, comme nous allons le voir, de «bonnes propriétés» en ce qui concerne la vérification et la modélisation.

Plutôt que d'étudier la composition dans sa totalité, nous allons utiliser la restriction suivante :

Définition 24 Composition parallèle //

Soient deux réseaux de Petri étiquetés $N_a = \langle P^a, T^a, \mathbf{Pre}^a, \mathbf{Post}^a, m_0^a, I_s^a, F^a, A^a, \Sigma_a^\epsilon, \mathcal{L}^a \rangle$ *et* $N_b = \langle P^b, T^b, \mathbf{Pre}^b, \mathbf{Post}^b, m_0^b, I_s^b, F^b, A^b, \Sigma_b^\epsilon, \mathcal{L}^b \rangle$.

Si $(\forall p \in P^a)(\forall p' \in P^b)(\mathcal{L}^a(p) \neq^\epsilon \mathcal{L}^b(p'))$*, alors la composition* $N_a \odot N_b$ *pourra être notée par* $N_a // N_b$*. L'opérateur // est dit opérateur de* composition parallèle*.*

L'opérateur // n'est donc applicable que lorsque seules des transitions sont fusionnées par l'opération de composition, c'est-à-dire lorsqu'il n'y a pas d'étiquettes de places concordantes entre les deux réseaux participant à la composition.

Maintenant que le cadre est posé, étudions donc le comportement de // face à l'exemple suivant :

(a) Un composant C_1 (b) Un composant C_2

Fig. 3.13 – Deux composants que nous souhaitons composer par //

Nous souhaitons composer les deux composants de la figure 3.13 en fusionnant leur seule transition possédant une étiquette commune : t_2 pour le composant C_1 et t_1 pour le composant C_2. Le résultat de la composition $C_1 // C_2$ est donné par le réseau de la figure 3.14.

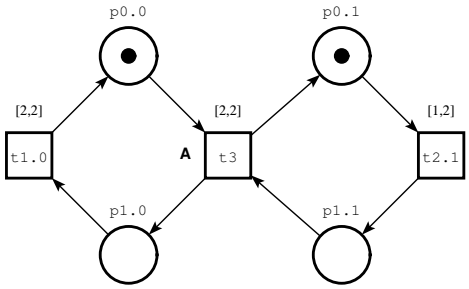


Fig. 3.14 – Résultat de la composition $C_1//C_2$

La transition *t3* est le résultat de la fusion de t_2 de C_1 avec t_1 de C_2. Son intervalle de tir est obtenu par l'intersection des intervalles des deux transitions fusionnées, le résultat est donc la valeur 2 (c'est-à-dire l'intervalle $[2, 2]$). Elle est tirable après le tir de *t2.1*, c'est-à-dire après une durée totale comprise dans l'intervalle $[3, 4]$, en comptant le délai ($[1, 2]$) préalable au tir de *t2.1* et celui ($[2, 2]$) préalable à son propre tir.

Alors que, dans le cas atemporel, la composition parallèle permet le codage d'un rendez-vous, ici nous voyons que le rendez-vous est temporellement forcé. En effet, nous avons composé deux transitions qui, au moins initialement, ne se "rencontrent" pas (ne partagent pas d'instants pendant lesquels elles sont toutes deux tirables). C'est en ce sens que nous considérons que cette composition donne des systèmes qui ne respectent pas (ou mal) les composants dont ils sont issus.

Le comportement du réseau $C_2//C_1$ de la figure 3.14 est donné par la figure 3.15.

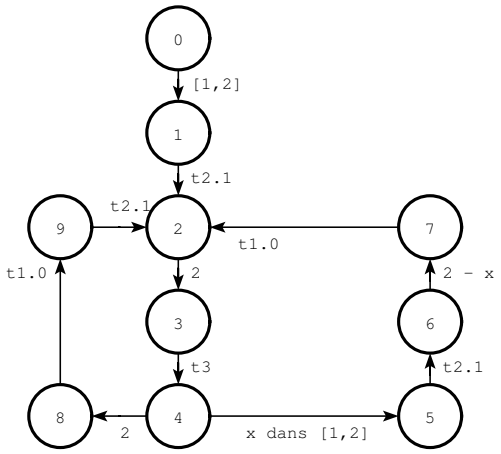


Fig. 3.15 – Comportement de $C_1//C_2$

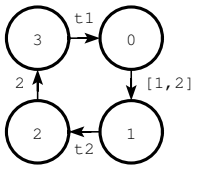


Fig. 3.16 – Comportement de C_1 et C_2

Si l'on ne souhaite observer que les évènements liés au composant C_1, on ne doit observer que t_3 et $t_1.0$, qui sont respectivement les évènements t_2 et t_1 de C_1. Le comportement de $C_1//C_2$ lorsqu'on n'observe que les actions de C_1 est donc celui de la figure 3.17.

De la même façon, on peut n'observer que les évènements du composant C_2 dans la composition $C_1//C_2$: le résultat de cette observation est donné par la figure 3.18. Ces deux observations sont à mettre en regard du comportement des composants C_1 et C_2 qui est donné par la figure 3.16.

Il apparaît clairement que l'activité d'un des composants est visible par l'autre à cause du temps mis pour effectuer une action. Il est donc impossible, si l'on utilise // sans précaution, de faire un pronostic sur le comportement de la composition en ne connaissant qu'un seul composant. De même, lors de la modélisation, il est primordial de connaître tous les composants pour modéliser un système composé correct. Or cela est contre toute intuition de la modélisation incrémentale, par laquelle un composant est une boite noire dont on ne présuppose que la connaissance de son interface.

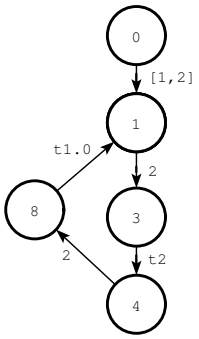


Fig. 3.17 – évènements de C_1 dans $C_1//C_2$

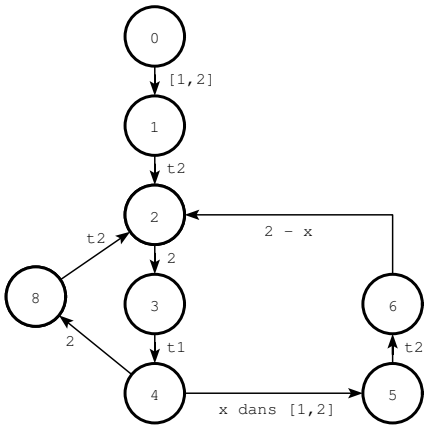


Fig. 3.18 – évènements de C_2 dans $C_1//C_2$

Pour améliorer cette situation, nous allons définir les réseaux de Petri composables, les seuls aptes à ne pas subir de distorsion temporelle lors de leur composition.

Définition 25 (*TPN* composable) *Un* TPN *étiqueté* $N = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s, \Sigma^\epsilon, \mathcal{L}\rangle$ *est* composable *lorsque* $(\forall t \in T)(\mathcal{L}(t) \neq \epsilon \Rightarrow I_s(t) = [0, \infty[)$, i.e. *toutes ses transitions étiquetées portent l'intervalle trivial* $[0, \infty[$.

Théorème 3 (Compositionnalité restreinte de // pour les *ipTPN*)
Soit $\lceil n \rceil$ *le* TTS *donnant le comportement du réseau* ipTPN n,
$N_a = \langle P^a, T^a, \mathbf{Pre}^a, \mathbf{Post}^a, m_0^a, I_s^a, F^a, A^a, \Sigma_a^\epsilon, \mathcal{L}^a\rangle$,
$N_b = \langle P^b, T^b, \mathbf{Pre}^b, \mathbf{Post}^b, m_0^b, I_s^b, F^b, A^b, \Sigma_b^\epsilon, \mathcal{L}^b\rangle$ *et*
$N = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s, F, A, \Sigma^\epsilon, \mathcal{L}\rangle$ *leur produit par //,*
$T^{i\backslash j}$ *l'ensemble de transitions étiquetées sur* $\Sigma_i^\epsilon \setminus \Sigma_j$,
Alors, $(\forall t \in T_{a\backslash b})(I_s^a(t) = [0, \infty[) \wedge (\forall t \in T_{b\backslash a})(I_s^b(t) = [0, \infty[)$
$\Rightarrow \lceil N_a//N_b \rceil = \lceil N_a \rceil \parallel \lceil N_b \rceil$

Le produit $\parallel$ de *TTS* a été présenté dans le chapitre précédent (définition 2).

Preuve La preuve de ce théorème est tout à fait analogue à celle donnée dans [BPV06]. □

La compositionnalité restreinte est une propriété intéressante car elle permet de connaître le comportement d'une composition de réseaux $\lceil N_a//N_b \rceil$ par le produit des *TTS* $\lceil N_a \rceil|||\lceil N_b \rceil$. Or le produit de *TTS* ne prend pas en compte les informations temporelles, ce qui signifie que les délais de l'un et l'autre des composants s'écoulent de manière indépendante. Pour faire le lien avec l'observation des actions d'un composant que nous avons faites un peu plus tôt, lorsqu'on n'observe que les actions de N_a dans $\lceil N_a \rceil|||\lceil N_b \rceil$, il n'est pas possible d'observer les délais de N_b.

Il est très facile de savoir quel *TPN* est composable. Le problème ici est de fournir une méthode systématique permettant d'obtenir un réseau composable à partir d'un réseau qui ne l'est pas. De plus, si un réseau doit être transformé de manière à devenir composable, nous souhaitons qu'il soit au minimum faiblement temporellement bisimilaire. Ce travail de conversion n'est pas possible de manière endogène et générale. Néanmoins une telle transformation est possible si les bornes supérieures des intervalles des transitions de l'interface du composant ne sont pas des bornes strictes [PBV09]. Cette transformation est très coûteuse.

Si la transformation endogène d'un composant *TPN* n'est pas toujours possible, une transformation exogène, en passant par les *ipTPN*, est possible de manière générale. Le résultat de la transformation est très proche du réseau d'origine et produit un composant *fortement bisimilaire*.

Reprenons ligne par ligne l'algorithme 2 montrant comment passer d'un *TPN* quelconque à un *ipTPN* fortement temporellement bisimilaire et composable. La bijection B associe à chaque transition du réseau source une nouvelle transition dans le réseau résultat. En effet, la traduction opère en

Algorithme 2 : Transformation d'un *TPN* non composable en un *ipTPN* composable équivalent

Soit le *TPN* $R = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s, \Sigma^\epsilon, \mathcal{L}\rangle$ et son interface $\mathcal{I} = \{t \in T | \mathcal{L}(t) \neq \epsilon\}$. Soit $\mathcal{I}'$ un ensemble de lettres distinctes de T (et de P) et $|\mathcal{I}| = |\mathcal{I}'|$. On peut définir une bijection entre les éléments de $\mathcal{I}$ et de $\mathcal{I}'$. Soit $B : \mathcal{I} \to \mathcal{I}'$ cette bijection.
L'*ipTPN* $R' = \langle P, T \cup \mathcal{I}', \mathbf{Pre}', \mathbf{Post}', m_0, I'_s, \multimap, \bullet\!\!-\!\!\circ, \Sigma, \mathcal{L}'\rangle$ suivant est l'équivalent composable de R :
a) $(\forall t \in T)(\mathcal{L}'(t) = \mathcal{L}(t))$
b) $(\forall t \in \mathcal{I})(\mathcal{L}'(B(t)) = \epsilon)$
c) $(\forall t \in \mathcal{I})(\mathbf{Pre}'(B(t)) = \mathbf{Pre}(t))$
d) $(\forall t \in T)(\mathbf{Pre}'(t) = \mathbf{Pre}(t) \wedge \mathbf{Post}'(t) = \mathbf{Post}(t))$
e) $(\forall t \in \mathcal{I})(B(t) \bullet\!\!-\!\!\circ\, t)$
f) $(\forall t \in \mathcal{I})(B(t) \multimap B(t))$
g) $(\forall t \in T \setminus \mathcal{I})(I'_s(t) = I_s(t))$
h) $(\forall t \in \mathcal{I})(I'_s(t) = [0, \infty[\wedge I'_s(B(t)) = I_s(t))$

récupérant l'ancien réseau, en le modifiant légèrement et en y ajoutant de nouvelles transitions. Les transitions de l'interface sont toutes transformées de telle façon qu'elles ne possèdent plus qu'une information temporelle triviale et les nouvelles, non composables et ne faisant pas partie de l'interface, portent l'information précédemment liée aux transitions de l'interface $\mathcal{I}$.

Chaque transition du réseau source garde son étiquette dans le réseau destination (a). Les transitions nouvellement ajoutées, c'est-à-dire les transitions de $\mathcal{I}'$, images par B de l'interface $\mathcal{I}$, portent toutes l'étiquette neutre ϵ (b). Ceci est indispensable pour s'assurer de la composabilité du nouveau réseau puisque ce sont ces transitions qui vont porter une information temporelle, à la place des transitions de $\mathcal{I}$ (h). Les transitions de $\mathcal{I}'$, nouvellement ajoutées, ont les mêmes préconditions que leurs homologues dans $\mathcal{I}$ (c), par contre elles n'ont pas gardé les postconditions : ce serait inutile vu que ces transitions de $\mathcal{I}'$ ne sont jamais tirables (f). Quant aux préconditions et postconditions des transitions du réseau initial, elles ne sont pas modifiées (d). Ne sont pas modifiées également les intervalles de tir des transitions n'appartenant pas à l'interface (g).

En appliquant cette transformation aux réseaux non composable C_1 et C_2 des figures 3.13(a) et 3.13(b), on obtient les réseaux des figures 3.19 et 3.20, qui, une fois composés selon la composition $C_1 \odot C_2$ donnent le réseau de la figure 3.21.

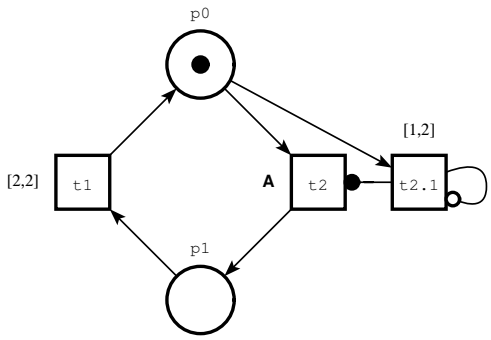


FIG. 3.19 – C_1, version *ipTPN* composable

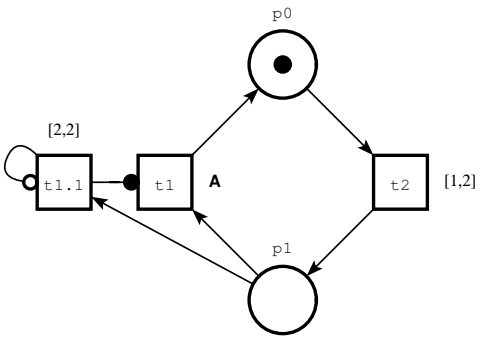


FIG. 3.20 – C_2, version *ipTPN* composable

A la différence de la composition $\odot$ des réseaux C_1 et C_2 non composables, dont le résultat est visible sur la figure 3.14, nous pouvons constater qu'ici la transition $t3$, résultat de la fusion des transitions étiquetées par A est contrainte non plus par un intervalle de tir, comme c'était le cas dans la composition précédente, mais par deux contraintes de permission. Ces deux contraintes sont contradictoires et empêchent la transition de tirer, mais également *empêchent le temps de progresser* : ni t_1, ni t_2 ne sont tirables (elles s'inhibent elles-mêmes), mais elles possèdent toutes deux une borne

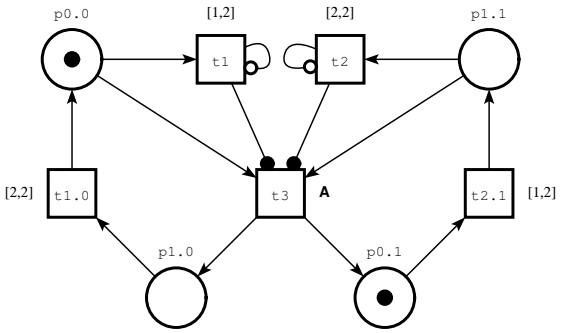


Fig. 3.21 – Résultat de la composition $C_1 \odot C_2$

supérieure, limitant la progression du temps au minimum de ces dernières. Puisque ce sont les seules transitions sensibilisées mise à part $t3$ (qui, rappelons-le, n'est pas tirable puisque les deux contraintes exercées par t_1 et t_2 sont contradictoires), le temps est condamné à se retrouver bloqué par la plus petite borne supérieure des intervalles de tir de t_1 et t_2. Ce blocage, nous semble-t-il est plus proche de ce que l'on attend d'une composition parallèle de systèmes temporels que la composition des *TPN* précédente : vouloir faire la composition comme nous le voulions n'était pas vraiment souhaitable et le blocage temporel est le résultat nous démontrant que cette composition est une mauvaise spécification si elle doit représenter un système réel.

Si par construction les *ipTPN* permettent plus de comportements «bizarres» — le blocage temporel est impossible avec les réseaux de Petri temporels — ils permettent dans le même temps plus d'expressivité et par conséquent plus de problématiques sont modélisables.

3.3.4 Les *ipTPN* pour coder les TA

Les automates temporisés sont une référence incontournable en ce qui concerne la spécification des systèmes temps réel. Il est intéressant de noter que la pratique de ce langage plutôt que celle des réseaux de Petri, n'a jamais eu d'argument définitif en sa faveur. La réciproque est également vraie. Une des raisons pouvant expliquer cette situation de *status quo* peut venir de ce que les contraintes temporelles sont manipulées suffisamment différemment dans l'un et l'autre des formalismes pour que l'on puisse trouver l'utilité de l'un et de l'autre selon telle ou telle spécification. Et cela en laissant de côté le fonctionnement atemporel sous-jacent, qui lui aussi est assez différent pour justifier une telle cohabitation.

L'introduction des deux nouvelles relations a eu pour résultat de donner des propriétés aux réseaux de Petri temporels qui initialement étaient réservés aux automates temporisées. Nous avons vu dans la section précédente que nous avions un formalisme composable et en conclusion nous avons constaté la possibilité nouvelle de spécifier des blocages temporels. La composabilité est présente nativement dans les automates temporisés : ce n'est pas la transition qui porte l'information temporelle, donc il n'y a pas de problèmes de composition par la fusion de transitions. En ce qui concerne les blocages temporels, ils sont apparus en même temps que les invariants de location, conditionnant la progression du temps à la validité de l'invariant de la location courante.

Ces deux faits nous poussent à comparer l'expressivité et la concision de l'un par rapport à l'autre.

La comparaison des deux formalismes des *TPN* et des TA est un sujet de recherche qui a fait l'objet de beaucoup de résultats (par exemple, [BCH$^+$05a],[BCH$^+$05b], [CR06], [BRH06]). Nous avons déjà noté que les travaux menant à la proposition de l'extension présentée dans ce chapitre a fait l'objet d'une étape préalable, à savoir la *relation de priorité* présentée dans [BPV06]. Cette étape nous a permis de jeter un nouveau pont entre les deux formalismes : en ajoutant une relation de priorité entre transitions — laquelle relation est très proche de la relation d'inhibition F — nous avons pu obtenir une traduction garantissant l'existence d'une relation de bisimulation temporisée faible. Cette traduction prend en compte les invariants non stricts.

Bien qu'il soit possible de traduire des TA en TPN étendus par la relation d'inhibition (ou de priorité), nous n'exposerons pas cette traduction dans cette thèse (le lecteur intéressé la trouvera dans [BPV06]). Cette traduction est en fait plus simple en utilisant les deux relations F et A et pour ne rien gâcher cette traduction-ci garantie l'existence d'une bisimulation temporisée *forte* (contre la bisimulation temporisée faible pour celle de [BPV06]). Les invariants ne sont pas codables de manière à préserver la bisimulation temporisée forte, mais le même codage que celui introduit dans [BPV06] est réutilisable.

Définition 26 Un automate temporisé *est un tuple* $\langle Q, q^0, X, \mathcal{C}, \Sigma, Tr\rangle$*, où :*
- Q *est un ensemble fini de locations*
- q^0 *est la location initiale*
- X *est un ensemble fini d'horloges*
- $\mathcal{C}$ *est un ensemble fini de contraintes. Une contrainte est un triplet* $\langle x, c, op\rangle$*, avec* $x \in X$*,* $c \in \mathbb{Q}^+$*,* $op \in \{<, \leqslant, \geqslant, >\}$*.*
- Σ *l'alphabet d'action (ne contient pas l'action silencieuse* ϵ*)*
- $Tr \subseteq Q \times 2^{\mathcal{C}(X)} \times \Sigma^\epsilon \times 2^X \times Q$ *l'ensemble des transitions*

Nous abrégerons $(q, g, a, r, q') \in Tr$ en $q \xrightarrow{g,a,r} q'$. L'ensemble de contraintes g est appelé la *garde* de la transition. L'ensemble r est l'ensemble des horloges remises à zéro lors du franchissement de la transition.

Algorithme 3 : Traduction d'un TA en *ipTPN*

Soit $\Gamma = \langle Q, q^0, X, \mathcal{C}, \Sigma, Tr\rangle$, un automate temporisé et un réseau de Petri temporel à inhibition/permission $\Delta = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s, F, A, \Sigma^\epsilon, \mathcal{L}\rangle$. Soit $\mathcal{B}_{tr} : Tr \to T_1$ une application bijective avec $T_1 \subseteq T$. Soit $\mathcal{B}_\mathcal{C} : \mathcal{C} \to T_2$ une application bijective avec $T_2 \subseteq T$ et $T_1 \cap T_2 = \emptyset$.

1) $P = Q \cup X \wedge T = \{\mathcal{B}_{tr}(t) | t \in Tr\} \uplus \{\mathcal{B}_\mathcal{C}(c) | c \in \mathcal{C}\}$
2) $(\forall q \in Q \setminus \{q^0\})(m_0(q) = 0) \wedge m_0(q^0) = 1$
3) $(\forall t \in Tr)(t = q \xrightarrow{g,a,r} q' \Rightarrow \mathbf{Pre}(q, \mathcal{B}_{tr}(t)) = 1 \wedge \mathbf{Post}(q', \mathcal{B}_{tr}(t)) = 1 \wedge \mathcal{L}(\mathcal{B}_{tr}(t)) = a \wedge I_s(\mathcal{B}_{tr}(t)) = [0, \infty[)$
4) $(\forall x \in X)(m_0(x) = 1)$
5) $(\forall t \in Tr)(t = q \xrightarrow{g,a,r} q' \Rightarrow (\forall x \in r)(\mathbf{Pre}(x, \mathcal{B}_{tr}(t)) = 1 \wedge \mathbf{Post}(x, \mathcal{B}_{tr}(t)) = 1))$
6) $(\forall t \in Tr)(t = q \xrightarrow{g,a,r} q' \Rightarrow (\forall c \in g)(c = \langle x, d, \prec\rangle \Rightarrow \mathbf{Pre}(x, \mathcal{B}_\mathcal{C}(c)) = 1$
   $\wedge\ (\prec \in \{<, \geqslant\} \Rightarrow I_s(\mathcal{B}_\mathcal{C}(c)) = [d, \infty[)$
   $\wedge\ (\prec \in \{\leqslant, >\} \Rightarrow I_s(\mathcal{B}_\mathcal{C}(c)) =]d, \infty[)$
   $\wedge\ \mathcal{B}_\mathcal{C}(c) \multimap \mathcal{B}_\mathcal{C}(c)$
   $\wedge\ (\prec \in \{<, \leqslant\} \Rightarrow \mathcal{B}_\mathcal{C}(c) \multimap \mathcal{B}_{tr}(t))$
   $\wedge\ (\prec \in \{>, \geqslant\} \Rightarrow \mathcal{B}_\mathcal{C}(c) \bullet\!\!- \mathcal{B}_{tr}(t)))$

Les transitions de l'automate Γ sont traduites par les transitions du réseau Δ images de $\mathcal{B}_{tr}$ tandis que les contraintes d'horloges de Γ sont traduites par les transitions de Δ images de $\mathcal{B}_\mathcal{C}$. Ces transitions ne sont pas tirables car elle s'inhibent elles-mêmes ($\mathcal{B}_\mathcal{C}(c) \multimap \mathcal{B}_\mathcal{C}(c)$). Les seules transitions tirables sont celles de l'ensemble $\{\mathcal{B}_{tr}(t) | t \in Tr\}$ des transitions traduisant celles de l'automate Γ. Il n'y en a pas d'autres car l'ensemble T des transitions de Δ est donné par l'union de ces deux ensembles (point 1).

Chaque location de Γ est directement traduite par une place de Δ. Comme la location initiale est q^0, la place correspondante est initialement marquée (point 2). Le point 3 décrit la traduction du squelette de l'automate. La seule précondition (resp. postcondition) de la transition $\mathcal{B}_{tr}(t)$ est la place représentant la location d'origine (resp. destination) de la transition t de l'automate.

Chaque horloge est codée par une place (point 4) initialement marquée et dont le jeton n'est jamais enlevé, sauf lors de la remise à zéro de l'horloge, auquel cas le jeton est enlevé et immédiatement remis

(point 5). Il y a par conséquent toujours un jeton dans ces places. Le point 6 gère les gardes, chacune d'elles étant codée par une transition jamais tirable (puisque $\mathcal{B}_{\mathcal{C}}(c) \multimap \mathcal{B}_{\mathcal{C}}(c)$), d'information temporelle ($I_s(\mathcal{B}_{\mathcal{C}}(c))$), liée à une transition tirable par la relation F ou A, selon la contrainte à coder.

Les transitions $\mathcal{B}_{tr}(t)$ ne portent pas d'information temporelle mais sont contraintes par inhibition/permission par les transitions $\mathcal{B}_{\mathcal{C}}(c)$ codant les contraintes c. Examinons maintenant ce codage :

$c = x < d : I_s(\mathcal{B}_{\mathcal{C}}(c)) = [d, \infty[$ et $\mathcal{B}_{\mathcal{C}}(c) \multimap \mathcal{B}_{tr}(t)$ signifie que $\mathcal{B}_{tr}(t)$ est tirable tant que $\mathcal{B}_{\mathcal{C}}(c)$ n'est pas T-sensibilisée. Autrement dit, $\mathcal{B}_{tr}(t)$ n'est tirable que d unités de temps après la dernière remise à zéro de $\mathcal{B}_{\mathcal{C}}(c)$. Parce que x est précondition de $\mathcal{B}_{\mathcal{C}}(c)$, $\mathcal{B}_{\mathcal{C}}(c)$ est T-sensibilisée dès que d unités de temps se sont écoulées.

$c = x \leqslant d : I_s(\mathcal{B}_{\mathcal{C}}(c)) =]d, \infty[$ et $\mathcal{B}_{\mathcal{C}}(c) \multimap \mathcal{B}_{tr}(t)$, signifie que $\mathcal{B}_{tr}(t)$ est tirable tant que $\mathcal{B}_{\mathcal{C}}(c)$ n'est pas T-sensibilisée, c'est à dire tant que $\mathcal{B}_{\mathcal{C}}(c)$ n'a pas dépassé d.

$c = x > d : I_s(\mathcal{B}_{\mathcal{C}}(c)) =]d, \infty[$ et $\mathcal{B}_{\mathcal{C}}(c) \multimap\!\!\bullet\ \mathcal{B}_{tr}(t)$, ce qui signifie que $\mathcal{B}_{tr}(t)$ n'est tirable que lorsque $\mathcal{B}_{\mathcal{C}}(c)$ est T-sensibilisée, ce qui signifie que $\mathcal{B}_{tr}(t)$ n'est tirable qu'après avoir été continûment sensibilisée pendant plus de d unités temps.

$c = x \geqslant d : I_s(\mathcal{B}_{\mathcal{C}}(c)) = [d, \infty[$ et $\mathcal{B}_{\mathcal{C}}(c) \multimap\!\!\bullet\ \mathcal{B}_{tr}(t)$, signifie que $\mathcal{B}_{tr}(t)$ est tirable dès que $\mathcal{B}_{\mathcal{C}}(c)$ est T-sensibilisée, c'est-à-dire dès que d unités de temps se sont écoulés depuis la dernière remise à zéro.

Ceci conclut la correction du codage des contraintes. Nous avons par ailleurs vu que les seules transitions tirables sont exactement celles traduisant les transitions de l'automate temporisé et finalement que les horloges encodées sont remises à zéro lors du tir des transitions traduisant celles de l'automate portant un *reset* d'horloges. Le comportement temporisé de l'automate temporisé sera donc nécessairement le même (au sens de l'égalité) et la bisimulation temporisée forte est assurée.

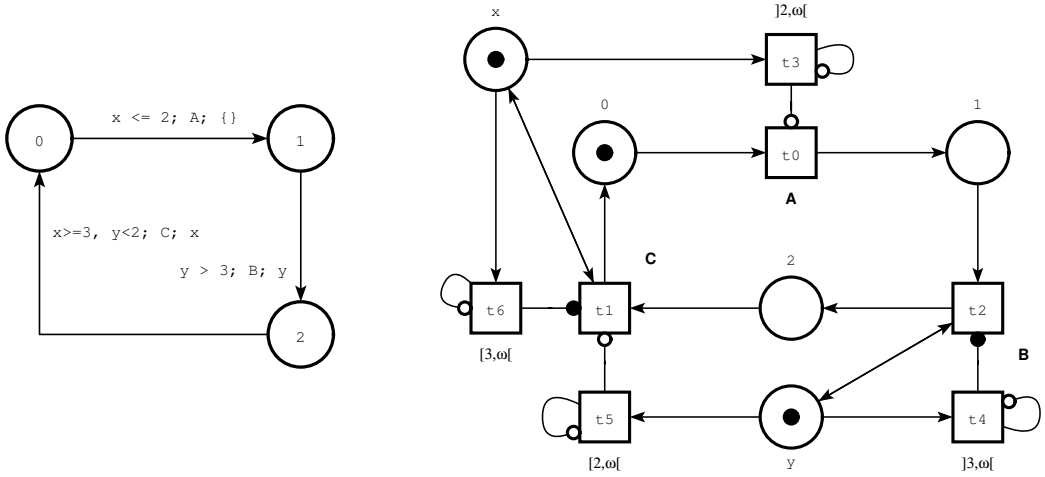


Fig. 3.22 – Un automate temporisé et sa traduction en *ipTPN*

Dans la figure 3.22 sont représentés un automate temporisé et sa traduction en *ipTPN*. La location initiale de l'automate est 0. Les transitions sont étiquetées selon le triplet (g, a, r), les gardes sont séparées par une virgule, les éléments du triplet par un point-virgule. L'ensemble vide est noté par $\{\}$. Cet automate fait donc dans l'ordre un A, puis un B, puis un C. Le premier A n'est produit que si $x \leqslant 2$, au-delà, l'automate est bloqué dans la location 0 où seul le temps peut diverger. Dans la location 1, un B est produit dès que $y > 3$ est vrai. L'horloge y est remise à zéro lors du franchissement de la transition. La transition faisant un C admet deux contraintes d'horloges $x \geqslant 3$ et $y < 2$ et remet à zéro l'horloge x lors de son franchissement.

Sur la figure 3.22, les places 0, 1 et 2 ainsi que les transitions t_0, t_1 et t_2 et les arcs les reliant forment la traduction du squelette atemporel de l'automate. Les contraintes d'horloges sont ensuite greffées. Pour x, la transition t_3 empêche t_0 de tirer après 2 unités de temps et la transition t_6 empêche t_1 de tirer avant que l'horloge interne de t_6 n'atteigne 3 unités de temps. Pour y, la transition t_4 empêche de tirer t_2 jusqu'à 3 unités de temps (3 inclus), alors que t_5 empêche de tirer t_1 au partir de 2 unités de temps. Les quatre transitions t_3, t_4, t_5 et t_6 possèdent une boucle inhibitrice qui interdit leur tir.

La traduction présentée dans cette section concerne les automates temporisés sans invariants. Les invariants non stricts peuvent être pris en compte et traités comme dans [BPV06]. Le traitement ne préserve plus la bisimilarité forte, toutefois, mais la traduction ainsi obtenue est globalement beaucoup plus compacte que celle de [BPV06].

3.4 Abstractions de l'espace d'états pour les *ipTPN*

La vérification par *model checking* a besoin d'une structure de graphe finie représentant le comportement du système. Nous avons déjà vu que les comportements temporels sont intrinsèquement en nombres infinis : il est donc nécessaire d'utiliser une abstraction finie de ces comportements. Une abstraction possède la propriété de préserver (ou non) certaines logiques, nous permettant ainsi de vérifier des propriétés du système concret directement sur l'abstraction.

Nous nous sommes concentré sur la préservation de la logique *LTL* par l'abstraction du graphe de classe dans le chapitre introduisant la vérification des réseaux de Petri temporels. L'ajout de deux nouvelles relations aux *TPN* ne change pas la problématique de représentation finie de l'espace d'états. Nous allons donc examiner ici comment cette abstraction peut être construite en présence de ces deux nouvelles relations, tout en gardant l'objectif de préservation de la logique *LTL*.

3.4.1 Graphe de classes (à temps) linéaire

La construction des classes d'états classique de [BM83], [BD91], appelée *LSCG* dans le chapitre précédent, vient de l'observation que, si deux classes sont équivalentes (rappel : leur marquage est identique et l'espace des solutions du domaine de tir est le même pour les deux classes) alors tout échéancier de tir tirable depuis un état de l'une des classes l'est également depuis un état de l'autre, et réciproquement.

Le *LSCG* est une abstraction efficace pour le *model checking* de formules *LTL*, car elle préserve les marquages et les traces maximales du réseau. Malheureusement, cette abstraction est trop faible pour préserver les effets de la relation d'inhibition. En effet, l'observation fondamentale, montrant que les ensembles d'états (les classes) sont équivalents lorsque l'union des solutions de contraintes de tirs des états (ainsi que de leurs marquages associés) sont les mêmes, n'est pas vraie en présence de cette relation d'inhibition, comme le montre l'exemple suivant.

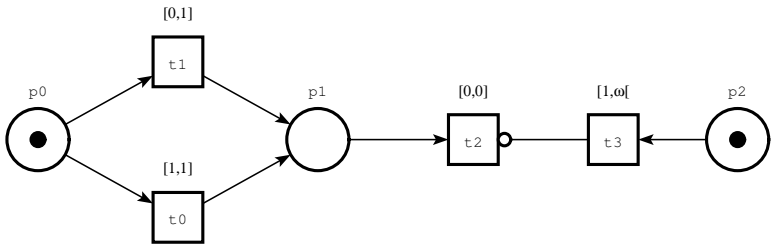


Fig. 3.23 – Un *ipTPN* qui ne peut être abstrait par un LSCG

Tirer l'une ou l'autre des transitions t_0 et t_1 dans le réseau de la figure 3.23 mène à la même classe du *LSCG*. Or, parce que la transition t_3 inhibe la transition t_2 et reste sensibilisée pendant le tir de t_0 ou de t_1, t_2 ne peut jamais tirer après t_0 mais peut être tirée après t_1 dans le cas où t_1 est tirée strictement avant l'instant 1.

Cet exemple montre que ni les marquages, ni les traces maximales, qui étaient préservées par le *LSCG*, ne le sont si l'on utilise la relation d'inhibition. La préservation des traces maximales est indispensable pour préserver les formules de la logique *LTL*. Cette abstraction n'est donc pas adaptée pour capturer le comportement des *ipTPN*.

Analysons plus en détails le problème. Si l'on génère le graphe du modèle en omettant la relation d'inhibition (et de permission, dans la suite de cette section nous ne préciserons plus cette relation : elle

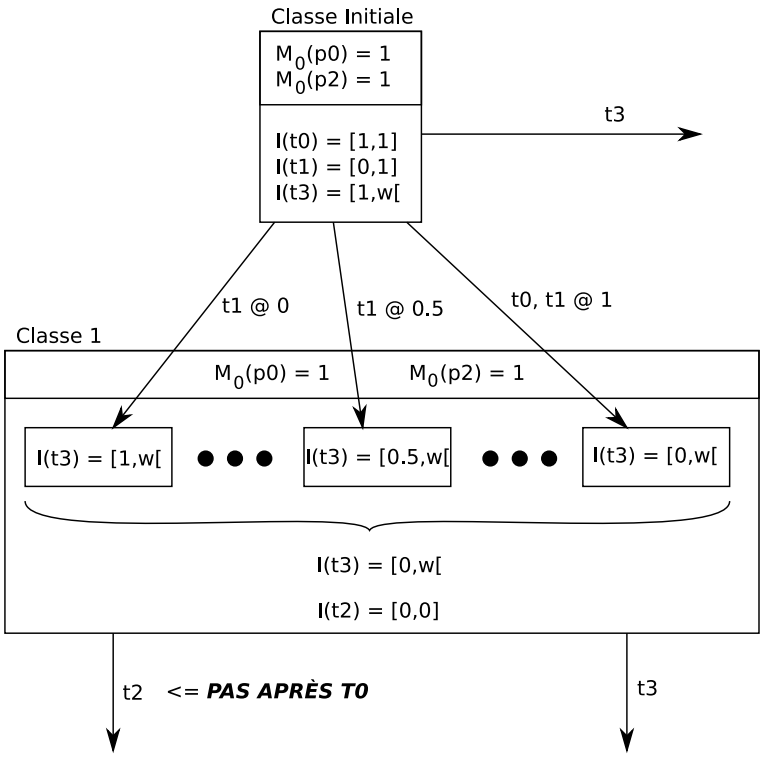


Fig. 3.24 – Le graphe des classes linéaire est inadapté aux *TPN* à inhibitions/permissions

va de pair avec son homologue inhibitrice), de la classe initiale on peut tirer t_0 et t_1 et aboutir dans la même classe 1. Aucune des transitions t_0 et t_1 n'est dans la relation d'inhibition, par conséquent, dans l'hypothèse où la construction du *LSCG* d'un modèle possédant une relation d'inhibition non-vide est possible, cette classe 1 est calculée de la même façon que dans le modèle sans relation d'inhibition. Or, comme nous l'avons vu ci-avant, dans le modèle où $t_3 \multimap t_2$, tirer t_0 ou t_1 n'autorise *pas* les mêmes séquences de tirs. C'est donc la classe 1 que nous allons étudier en détail.

Nous avons ici un cas typique d'abstraction trop grossière : deux classes qui devraient être distinctes pour notre problème sont amalgamées à cause d'une opération destructrice d'informations (destructrice en regard des objectifs de notre problème). Il est normal qu'une abstraction détruise de l'information, mais cette destruction est ici trop importante pour que notre but de préserver *LTL* soit atteignable.

Quand deux classes sont équivalentes alors leur domaine de tir est équivalent (leur marquage aussi, mais ce n'est pas la source du problème). Ce domaine détermine un ensemble d'états, c'est d'ailleurs ainsi que nous avons justifié la construction en classes : par le calcul de ce domaine. Le problème vient donc d'un domaine trop large.

Le calcul du domaine de tir utilise une opération d'union et une opération de décalage d'intervalle : l'intervalle temporel des transitions persistantes est décalé de la quantité de temps attendue avant le tir de la transition ; et comme il est possible d'avoir une infinité de valeurs possibles pour la progression temporelle (définissant un intervalle), on doit faire l'union des décalages des intervalles de tirs des transitions persistantes.

Du moment que l'opération de décalage d'un intervalle donné est injective (il est possible de «remonter» le résultat d'une opération de décalage), il est possible de retrouver comment a été construit chaque état de la classe. Il est dès lors possible de ne regrouper que les états à partir desquels on peut tirer les mêmes transitions (après avoir attendu le temps requis pour cela).

Théorème 4 *Soient $a \in \mathbb{N}$, $b \in \mathbb{N}$ et $c \in \mathbb{N}$. La sémantique des réseaux de Petri temporels impose que :*

$$[a,b] \dot{-} c = [a,b] \dot{-} d \Rightarrow c = d \tag{3.5}$$

Preuve Soit $[y, z]$ le résultat de l'opération $[a, b] \dot{-} c$. On doit vérifier que $(\forall c, d \in \mathbb{N}^2)([a, b] \dot{-} c = [a, b] \dot{-} d \Rightarrow c = d)$. On a :

$$\begin{array}{rcccl} y & = & a \dot{-} {}^1c & = & a \dot{-} d \\ z & = & b \dot{-} c & = & b \dot{-} d \end{array}$$ Si $z > 0$, alors $\dot{-} = -$ et donc $c = d$.

Si $a - c = a - d = 0$ (donc on a encore $\dot{-} = -$) alors $c = d$, mais a seul ne permet pas de déduire l'égalité $c = d$ de manière générale, car l'on ne peut rien dire dans le cas où $z = 0$: le $\dot{-}$ détruit de l'information.

Pour avoir l'information il faut donc espérer que b permet de retrouver l'information perdue par $a \dot{-} c$. Pour cela il faut que $\dot{-} = -$ c'est-à-dire, que $z > 0$ ou $z = 0$ *et* $b - c = b - d = 0$. Supposons que cela ne soit pas le cas, c'est-à-dire que l'on puisse avoir $b - z < 0$. Cela signifie que $b < z$. En terme de sémantique des réseaux de Petri, cela signifie que le temps a progressé au delà de la limite autorisé. Il est donc impossible d'avoir $b - c < 0$ et donc $\dot{-} = -$ et finalement $c = d$. □

A partir de ce résultat nous savons que l'opération de décalage garde suffisamment d'information pour que l'on sache exactement de quel état l'on vient, après application du décalage. Comme on l'a dit plus haut, ceci est suffisant pour savoir discriminer les états selon leur futur, même en présence de contraintes d'inhibitions.

D'où vient donc le problème que nous constatons ? Le lecteur aura peut-être déjà deviné : dans l'exemple fautif de la figure 3.23, la transition t_3 possède un intervalle *non-borné*. Or le théorème ci-dessus ne peut s'appliquer que si b est fini, sinon la preuve ne tient plus. En l'occurrence on ne sait pas retracer les évènements ayant conduit à la classe 1 : si on attend 1 unité de temps, le décalage $[1, \infty[\dot{-} 1$ donne l'intervalle $[0, \infty[$ (en considérant $\infty - x = \infty$), mais c'est également le cas pour les décalages $[x, \infty[\dot{-} 1$ avec $x \in [0, 1]$. Il est donc impossible de connaître l'état avant décalage en ne regardant que son résultat, c'est-à-dire qu'une fois cette opération de décalage appliquée sur des domaines de tir, il n'est plus possible de connaître l'état d'origine et par conséquent tous les états d'origines sont considérés comme équivalents, ce qui est clairement incorrect en présence de la relation d'inhibition.

Parmi les représentations adéquates de l'état d'un réseau, il en est une que nous n'avons pas encore abordée et qui va nous permettre de surmonter la difficulté de représentation des *TPN* avec relation d'inhibitions. Au lieu de repérer un état par son domaine de tir (et son marquage), il est également possible d'utiliser la quantité de temps qui s'est écoulée depuis la dernière nouvelle sensibilisation de la transition. Cette grandeur est appelée *horloge* de la transition.

Théorème 5 *Dans un réseau où tous les intervalles sont bornés, tous les états définis en terme d'intervalles de tir d'une classe admettent un et seul état correspondant défini en terme d'horloges.*

Ce théorème est une conséquence immédiate du théorème précédent : si $[a, b]$ est l'intervalle de tir statique, alors $[a, b] \dot{-} c$ est la valeur de l'intervalle de tir après c unités de temps et c est la valeur de l'horloge de la transition, c'est-à-dire que le domaine de tir $[a, b] \dot{-} c$ est équivalent au domaine d'horloge $[c, c]$.

Si le réseau admet un intervalle non-borné la correspondance est perdue : par exemple, l'intervalle $[0, \infty[$ ne peut dénoter qu'un seul et unique état (avec son marquage associé), alors qu'en terme d'horloges, il en dénote une infinité : $[0, \infty[\dot{-} x = [0, \infty[$ et c'est vrai pour n'importe quelle valeur d'horloge $x \in N$. Ces états sont tous *différents* en terme d'horloges mais identiques en terme d'intervalles de tirs.

Cette précision supplémentaire apportée par les horloges est suffisante pour notre problème de représentation du comportement.

3.4.2 Graphe de classes linéaires fortes

La construction LSCG précédente utilise la même notion d'états que celle utilisée dans la sémantique : un état est donné par le marquage courant et les intervalles de tir courants des transitions sensibilisées.

[1] $a \dot{-} c$ est un raccourci pour $[a, a] \dot{-} c$

Ainsi deux états qui possèdent le même intervalle de tir sont considérés équivalents (si par ailleurs il possèdent le même marquage bien sûr). Nous avons vu que cette équivalence pouvait entraîner des problèmes dans le cadre des relations d'inhibition/permission lorsque l'union des intervalles de tir vient se rajouter. Dans cette section nous présenterons une représentation alternative des états, plus précise, utilisant *l'horloge* des transitions au lieu des intervalles de tir. Le graphe qui est présenté ici s'appelle le graphe de classes linéaires fortes, abrégé SSCG.

Domaines d'horloges : A chaque état accessible, on peut associer une *fonction d'horloge* γ. La fonction γ associe, à chaque transition sensibilisée, le temps écoulé depuis sa dernière sensibilisation. Les fonctions d'horloges peuvent aussi être vues comme des vecteurs $\underline{\gamma}$ indexés par les transitions sensibilisées (c'est-à-dire que chaque transition représente une dimension et l'index-transition permet de récupérer la valeur de la composante de la dimension associée à la transition).

Dans la construction SSCG, une classe est représentée par un marquage et un système d'horloges, mais les classes représentent toujours des ensembles d'états comme définis par la sémantique (c'est-à-dire par des fonctions intervalles de tir). Un système d'horloges est un système d'inéquations sur ces horloges, qui peut être donné sous forme matricielle par l'application d'une matrice de coefficients G, au vecteur $\underline{\gamma}$ des termes de l'inéquation (c.-à-d. les horloges), laquelle application est ensuite comparée à un vecteur de constantes $\underline{g}$, soit au final le système d'inéquations $G\underline{\gamma} \leqslant \underline{g}$. L'ensemble d'états ayant le marquage m et le système d'horloge $Q = \{G\underline{\gamma} \leqslant \underline{g}\}$ est l'ensemble $\{(m, \overline{\Phi}(\underline{\gamma})) \mid \underline{\gamma} \in \langle Q \rangle\}$, où $\langle Q \rangle$ est l'ensemble solution de Q et le domaine de tir $\overline{\Phi}(\underline{\gamma})$ est l'ensemble solution dans $\underline{\phi}$ du système :

$$\underline{0} \leqslant \underline{\phi}\ ,\ \underline{e} \leqslant \underline{\phi} + \underline{\gamma} \leqslant \underline{l} \ \text{ avec } \ \underline{e}_k = \downarrow \mathbf{I}_s(k) \text{ et } \underline{l}_k = \uparrow \mathbf{I}_s(k)$$

En mettant pour l'instant de côté le marquage, chaque vecteur d'horloges représente un état, mais, à moins que les intervalles statiques de toutes les transitions soient bornés, des vecteurs d'horloges différents peuvent représenter le même état, et des systèmes d'horloges avec des ensembles solution différents (souvent une infinité) peuvent représenter le même ensemble d'états. Pour cette raison, nous introduisons l'équivalence $\equiv$:

Définition 27 *Etant donné* $c = (m, Q = \{G\underline{\gamma} \leqslant \underline{g}\})$ *et* $c' = (m', Q' = \{G'\underline{\gamma}' \leqslant \underline{g}'\})$, $c \equiv c'$ *ssi* $m = m'$ *et les systèmes d'horloges* Q *et* Q' *représentent les même ensembles d'états.*

L'équivalence $\equiv$ est décidable, des méthodes efficaces pour la vérifier sont présentées dans [BV03] et [Had06], elles s'appuient sur la relaxation de systèmes d'horloges. Quand les intervalles statiques de toutes les transitions sont bornés, $\equiv$ est l'égalité des ensembles solutions des systèmes d'horloges.

Les relaxations permettant d'obtenir dans tous les cas un graphe de comportement fini réintroduisent les intervalles non-bornés qui nous avaient gênés dans la section précédente. Dans le cas des domaines d'horloges, les intervalles ne deviennent non-bornés que suite à une relaxation et une relaxation ne concerne que la partie telle que $\underline{\gamma}_t \geqslant \downarrow I_s(t)$, c'est-à-dire que les états équivalents par la relaxation (c.-à-d. ceux qui possèdent le même intervalle de tir) sont équivalents en ce qui concerne les possibilités d'actions discrètes futures. Ceci ne change pas dans le cadre de l'introduction des relations d'inhibitions et de permissions puisqu'elles ne sont discriminantes que par rapport à la borne $\downarrow I_s(t)$.

Construction du SSCG : Les classes d'états fortes sont représentées par un marquage m et un système $Q = \{G\underline{\gamma} \leqslant \underline{g}\}$ décrivant un domaine d'horloge pour les transitions sensibilisées. La variable d'horloge $\underline{\gamma}_i$ est associée à la $i^{ème}$ transition sensibilisée par m.

La première étape du calcul d'une classe successeur, lorsque l'on construit le SSCG, est de déterminer quelles transitions sont tirables depuis la classe courante. En l'absence de relations d'inhibition/permission, la transition t est tirable depuis un état de la classe (m, Q) ssi il existe un délai $\theta \in \mathbb{R}^+$ tel que, augmenté de θ, les horloges de toutes les transitions sensibilisées ne dépassent pas la borne supérieure de leur intervalle statique respectif, et dans cet intervalle, pour la transition t :

(a1) $\theta \geqslant 0$

(a2) $\theta + \underline{\gamma}_t \in \mathbf{I}_s(t)$

(a3) $(\forall i \neq t)(m \geqslant \mathbf{Pre}(i) \Rightarrow \theta + \underline{\gamma}_i \leqslant \uparrow \mathbf{I}_s(i))$

En présence des relations de permission/inhibition, une quatrième et une cinquième condition doivent être ajoutées, certifiant qu'aucune des transitions sensibilisées inhibant t n'est T-sensibilisée à θ (4a) mais que toutes les transitions permettant le tir de t le sont.

(a4) $(\forall i)(m \geqslant \mathbf{Pre}(i) \wedge i \multimap t \Rightarrow \theta + \underline{\gamma}_i \notin \mathbf{I}_s(i))$

(a5) $(\forall i)(m \geqslant \mathbf{Pre}(i) \wedge i \bullet\!\!- t \Rightarrow \theta + \underline{\gamma}_i \in \mathbf{I}_s(i))$

$\theta + \underline{\gamma}_i \notin \mathbf{I}_s(i)$ est vraie ssi $\theta + \underline{\gamma}_i > \uparrow \mathbf{I}_s(i)$ ou $\theta + \underline{\gamma}_i < \downarrow \mathbf{I}_s(i)$, mais le premier cas ne peut jamais se produire puisque cela contredit la condition (a3).

Le SSCG d'un *ipTPN* est construit de la même manière que celui d'un *TPN*, en ajoutant simplement les conditions de tir comme nous venons de le voir (voir algorithme 4).

Algorithme 4 : Calcul du graphe de classes d'états fortes avec priorités

- $R_\epsilon = (m_0, \{0 \leqslant \underline{\gamma}_t \leqslant 0 \mid m_0 \geqslant \mathbf{Pre}(t)\})$
- Si σ est tirable et $R_\sigma = (m, Q)$ alors $\sigma.t$ est tirable ssi
  - (i) $m \geqslant \mathbf{Pre}(t)$
  - (ii) Q auquel on ajoute :
    $\theta \geqslant 0$, $\theta \geqslant \downarrow \mathbf{I}_s(t) - \underline{\gamma}_t$
    $\{\theta \leqslant \uparrow \mathbf{I}_s(i) - \underline{\gamma}_i \mid m \geqslant \mathbf{Pre}(i)\}$
    $\{\theta < \downarrow \mathbf{I}_s(j) - \underline{\gamma}_j \mid m \geqslant \mathbf{Pre}(j) \wedge j \multimap t\}$
    $\{\theta \geqslant \downarrow \mathbf{I}_s(j) - \underline{\gamma}_j \mid m \geqslant \mathbf{Pre}(j) \wedge j \bullet\!\!- t\}$
    est consistant.
- Si $\sigma.t$ est tirable alors $R_{\sigma.t} = (m', Q')$ est calculé à partir de $R_\sigma = (m, Q)$:
  m' = m - **Pre**(t) + **Post**(t)
  Q' obtenu par :
  - (a) Une nouvelle variable θ est introduite, contrainte par (ii) ci-dessus ;
  - (b) $\forall k \in \mathbf{T} : m' \geqslant \mathbf{Pre}(k)$, introduire une nouvelle variable $\underline{\gamma'}_k$, telle que :
    $\underline{\gamma'}_k = \underline{\gamma}_k + \theta$ si $k \neq t \wedge m - \mathbf{Pre}(t) \geqslant \mathbf{Pre}(k)$
    $0 \leqslant \underline{\gamma'}_k \leqslant 0$ sinon
  - (c) Les variables $\underline{\gamma}$ et θ sont éliminées.

La variable temporaire θ représente les délais possibles après lesquels t peut tirer. Il y a un arc portant le label t entre R_σ et c ssi $c \equiv R_{\sigma.t}$.

Comme pour les réseaux de Petri temporels, le SSCG d'un *ipTPN* est fini ssi le réseau est borné. Les systèmes d'horloges sont des systèmes de différences, pour lesquels des formes canoniques peuvent être calculées avec une complexité de temps $\mathcal{O}(n^2)$, si l'on suit l'observation de [Rok93] selon laquelle ils peuvent être construis de manière incrémentale. Le SSCG préserve les traces du réseau et permet de décider l'accessibilité des états [BV07].

Si l'on compare aux méthodes proposées pour le *model checking* des automates temporisés à priorités [LHHC05] [DHLP06], l'algorithme 4 ne nécessite pas la soustraction de DBM coûteuse ($\mathcal{O}(n^4)$) obligatoire pour ces modèles. Comme cela a été expliqué, cela vient du fait que le passage du temps dans les *ipTPN* est borné par la plus petite des bornes supérieures des transitions sensibilisées, quelle que soit la relation d'inhibition.

3.5 Conclusions

L'extension proposée dans ce chapitre apporte une contribution dans la concision d'expression et la modularité de spécification des réseaux de Petri. La concision peut sembler un aspect anecdotique de la spécification, surtout lorsque le modèle est de bas niveau et souvent destiné à être une cible de traduction automatique. Mais derrière cet aspect pratique se cache également un aspect plus problématique : si un système peut être spécifié avec moins de transitions, cela aura un impact sur le graphe de comportement : moins de transitions "inutiles" c'est un graphe de comportement plus

réduit. Bien sûr, le problème de l'explosion combinatoire ne va pas s'en trouvé miraculeusement résolu, mais c'est une aide qu'il convient de ne pas négliger.

L'une des facettes qui vont être exploré dans la suite de cette thèse est la faculté de composition permettant de décomposer une problématique, ici la spécification des systèmes de tâches temps réel, en autant de parties que nécessaire. Cette compositionnalité est un gain en terme de temps énorme : là ou un système devait être conçu d'un bloc, souvent avec des va-et-vient implémentation-débuggage intrinsèque à une modélisation en bloc, ici les méthodologies incrémentales peuvent s'appliquer. Un système peut être conçu, vérifié, et se voir composer à un autre par une simple opération. L'ajout de motifs-composants devient ainsi bien plus aisé.

Du point de vue de la vérification, la compositionnalité est une arme à double tranchant : si elle permet la vérification d'un composant avant même d'avoir conçu tout le système, elle introduit aussi, de fait, une nouvelle complexité. En effet, dans le cas d'un système clos, il n'y a pas à se préoccuper de l'environnement, puisque celui-ci n'interagit alors pas avec le système. Dans le cas d'un composant cela n'est plus vrai. Un composant interagit avec son environnement par définition. La vérification d'un composant se fait donc selon un environnement donné, et plus la donnée de cet environnement est lâche, plus son comportement est volumineux. Et plus un environnement est restreint ... plus il ressemblera au système monolithique dont nous tentons de nous défaire. La compositionnalité et la vérification forment ainsi une dialectique complexe intéressante à étudier, mais que nous n'avons pas abordé.

Chapitre 4

Le langage Pola

4.1 Introduction

4.1.1 Objectif

Lors de l'étude du formalisme des réseaux *ipTPN*, il s'est posé la question de leur adéquation à la modélisation des systèmes temps réel. Il nous paraît clair que ce formalisme est plus souple que les *TPN* et semble donc plus «adapté» à la modélisation des systèmes temps réel. Néanmoins cette perception est très subjective et n'apporte par d'informations réelles. Nous avons donc voulu appréhender de plus près comment cette adéquation pouvait être exprimée.

L'adéquation peut se comprendre comme la définition d'un périmètre des systèmes de tâches temps réel que l'on pourrait exprimer à l'aide des *ipTPN*.

Les concepts du domaine des systèmes temps réel étant très éloignés sémantiquement de ceux présents dans le formalisme des *ipTPN*, la définition de ce périmètre s'en trouve grandement complexifiée.

Le meilleur moyen d'être sûr qu'un langage est adapté à la modélisation d'un domaine est de fournir des primitives correspondants le plus fidèlement possible aux concepts utilisés dans le domaine. Un langage fournissant de telles primitives décrit «trivialement» le domaine, c'est-à-dire que sa lecture ne devrait pas impliquer d'apprentissage pour les experts du domaine. Les langages de ce type sont appelés *langages spécifiques à un domaine* (*domain specific languages* ou DSL).

Si la sémantique d'un tel DSL peut être donnée par le formalisme des *ipTPN*, alors, par «transitivité», on est assuré que ce dernier est capable de modéliser les systèmes exprimables à l'aide du DSL.

L'expressivité du langage n'est donc pas un impératif dès lors qu'on le souhaite le plus proche possible du domaine. D'autre part, l'objectif de vérification imposé par le formalisme des *ipTPN* implique une limitation stricte du domaine que l'on peut modéliser.

C'est cette approche que nous allons suivre dans les deux prochains chapitres : dans ce chapitre nous définirons le langage lui-même, tandis que dans le chapitre suivant, nous donnerons sa sémantique par un réseau *ipTPN*.

Pour finir, nous souhaitons noter un des aspects qui est venu «gratuitement» se greffer à cette étude. Notre motivation pour la conception d'un DSL est plutôt marginale : même si cerner les caractéristiques d'un domaine par un langage peut être *a posteriori* partagés par tout usage de DSL, elle n'en est souvent pas l'objectif principal. La plupart du temps, l'objectif de conception *a priori* d'un DSL vient de ce que l'utilisation de l'outillage informatique est jugée trop complexe pour des personnes non-expertes dans l'utilisation des langages informatiques, mais néanmoins expertes dans leurs domaines. Il y a ainsi un fossé à franchir pour appliquer cette expertise, étrangère à celles des langages informatiques, à une méthodologie nécessitant l'utilisation de ces langages. C'est un des principaux arguments en faveur des DSL : mettre à la portée de l'expert un langage permettant d'exprimer le plus naturellement possible les problèmes liés à son domaine. Seule une distance sémantique la plus réduite possible entre la sémantique du langage et celle du domaine peut apporter une réponse satisfaisante.

Ainsi, même si notre objectif premier n'était pas de concevoir un langage «pratique» pour un concepteur de système temps réel, étant donné que les objectifs convergent, notre langage dédié sera également utilisable facilement par des experts du domaine. Il est particulièrement intéressant de constater qu'une fois la sémantique du langage donnée en terme d'*ipTPN*, il est alors possible de concevoir une chaîne vérification partant du langage spécifique, cachant les aspects compliqués par une traduction automatique et donnant le résultat de vérifications pré-calibrées, sans que l'utilisateur n'ait eu autre chose à connaître que le langage dédié lui-même. Un prototype de cette chaîne de vérification a été conçu et des résultats d'expérimentations issus de son utilisation sont présentés au chapitre 6.

4.1.2 Un langage spécifique à un domaine

Une définition des langages spécifiques est toujours sujette à caution. En voici néanmoins une se basant sur celles données dans [vDKV00] et [Spi01] :

> *Un langage spécifique à un domaine (DSL en abrégé, pour* Domain Specific Language*) est un langage formel offrant un pouvoir expressif se focalisant sur, et généralement restreint à, un domaine de problèmes particuliers ; au lieu d'être général, il capture précisément (ou tente de capturer) la sémantique du domaine.*

Ces langages ne sont pas récents [Lan66] et sont même couramment utilisés dans des systèmes d'exploitations tels que Unix, parfois sous le nom de mini-langages [Ray03].

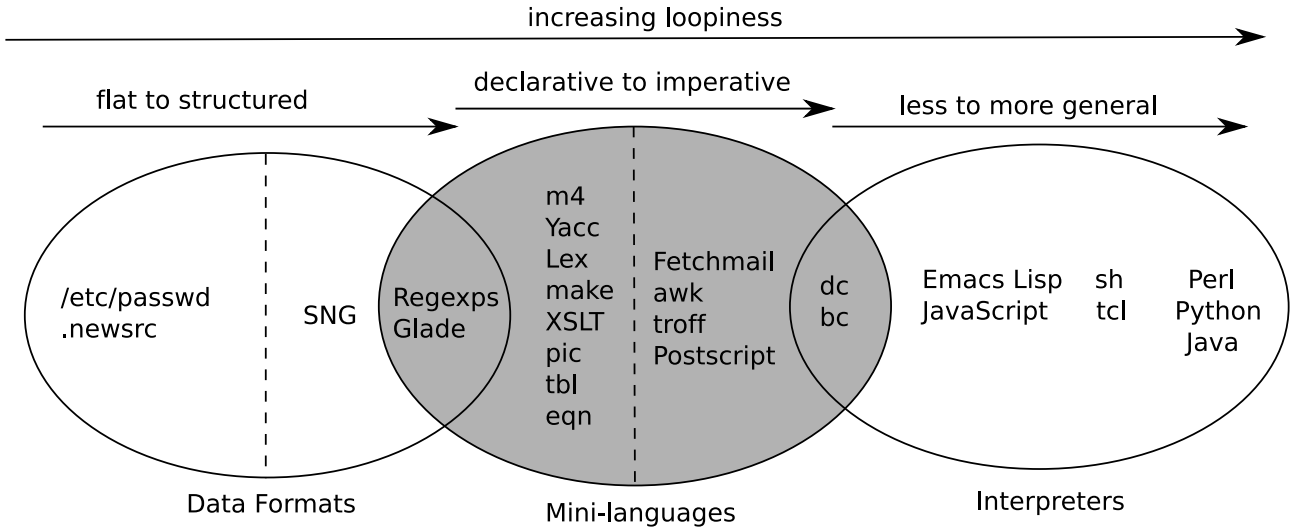


Fig. 4.1 – Une taxonomie des «mini-langages» d'Unix proposée par Eric S. Raymond [Ray03]

La taxonomie proposée dans [Ray03] pour les «mini-langages» d'Unix nous semble apporter une bonne vision d'ensemble de ce que sont les langages spécifiques à un domaine, dans leur généralité. L'auteur s'efforce par cette taxonomie de définir ce qu'il entend par «mini-langage», mais de notre point de vue, tous les langages présent dans cette taxonomie peuvent plus ou moins prétendre à un usage spécifique. Ainsi un DSL peut être structuré ou plat, et avoir un caractère plus ou moins déclaratif ou être plus ou moins d'usage général. Parce qu'il se concentre sur les langages des outils Unix, l'auteur ordonne ces caractéristiques (selon qu'ils possèdent plus ou moins de structures itératives (*loopiness*)), ce qui ne nous semble par contre pas correspondre à une généralité pour les DSL : ces trois critères énoncés précédemment peuvent se retrouver de manière indépendante.

4.1.3 Systèmes temps réel

Les systèmes temps réel peuvent avoir des réalisations très diverses et sont de ce fait difficilement caractérisables par une définition concise. Néanmoins certaines caractéristiques se dégagent permettant de parler de «systèmes temps réel».

D'une manière générale, un système temps réel fonctionne sur le principe ré-entrant des systèmes de contrôle-commande : c'est un système réagissant à un environnement en effectuant des acquisitions périodiques et en traitant les informations fournies par les capteurs d'acquisitions de manière à commander des actionneurs interagissant avec l'environnement.

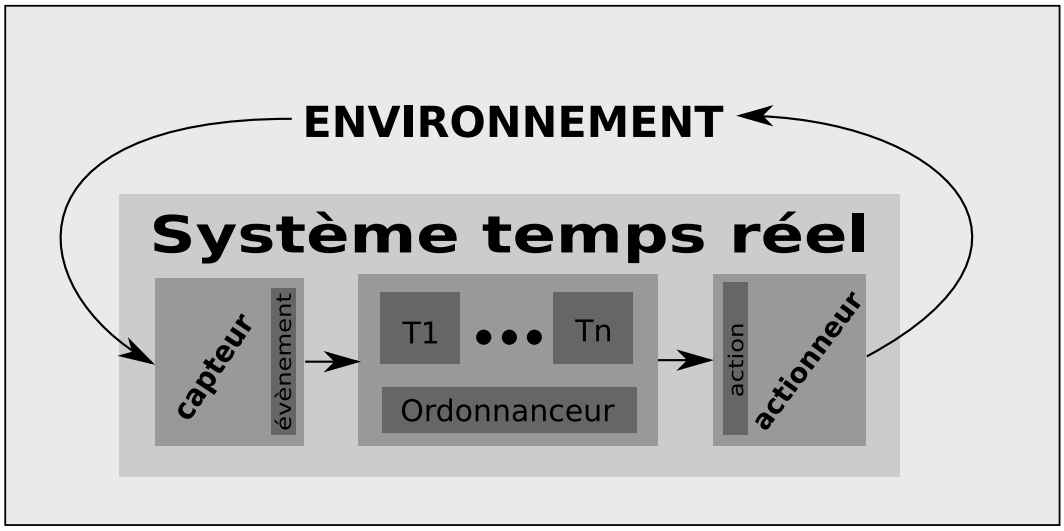


Fig. 4.2 – Un système temps réel

L'environnement évoluant avec le temps, il est nécessaire qu'un système temps réel ait conscience de cette évolution temporelle. Pour son bon fonctionnement le système doit utiliser les *mêmes* données temporelles que celles utilisées par son environnement. C'est le domaine temporel sur lequel ces données évoluent qui est qualifié de temps réel. Ce temps que l'on qualifie de réel est donc relatif à l'environnement dans lequel la contrainte s'exerce.

Ainsi, la fonctionnalité d'un système temps réel n'est donc pas seulement caractérisée par une réponse correcte aux stimuli de l'environnement, mais il doit également garantir une réponse en un temps déterminé. C'est donc ce caractère hybride qui caractérise le mieux et de façon générale un système temps réel : il doit être correct dans ses fonctions logiques mais également précis dans ses instants de réponse.

Avant de continuer l'exploration de ces systèmes, il est important de préciser que la correction et la fiabilité des systèmes temps réel ne sont pas juste une histoire de vitesse [Sta88]. Un tel système doit être suffisamment rapide pour répondre en temps requis, c'est là qu'intervient le paramètre vitesse. Si un système était constitué d'une seule fonction alors ce paramètre de vitesse serait suffisant pour décréter qu'un système satisfait ses contraintes temporelles. Mais dès lors que le temps d'exécution est partagé par les différentes tâches du système, la rapidité ne suffit plus : un traitement ponctuel nécessitant un temps de réponse très court peut avoir à interrompre une tâche n'ayant pas une contrainte temporelle aussi forte, la tâche interrompue étant redémarrée dès que l'exécution de la tâche nécessitant un temps de réponse court est terminée.

Parmi les exemples de systèmes temps réel, nous pouvons citer : la marche d'un robot, la vidéo par internet (ou tout autre type de flux), le contrôle de température d'une centrale nucléaire ou bien encore le contrôle de trajectoire d'un avion ou d'un satellite. Il est même possible de considérer qu'un logiciel de calcul de fiche de paie est un système temps réel : les conséquences d'un retard dans le paiement des salaires pouvant être critique.

Dans les deux premiers exemples, il n'est pas forcément très grave de rater des échéances : le robot peut avoir du mal à marcher, s'il peut se rattraper, l'erreur commise n'en reste que gênante ; de même pour la vidéo. Si l'image d'une vidéo s'arrête brusquement pendant un dixième de seconde, l'utilisateur va bien sûr être perturbé et gêné dans son visionnage, mais l'objectif principal, qui est de pouvoir comprendre la vidéo reste atteignable. Nous avons ici des systèmes acceptant une dégradation de la qualité de leur fonctionnement.

Par contre, le contrôle température d'une centrale ou le contrôle de trajectoire d'un avion sont bien plus critiques, autant du point de vue évident de la sécurité humaine, que de celui de la sécurité du système lui-même. Même si à court terme la vie de personnes n'est pas engagée, à terme, le bon fonctionnement du système peut être empêché par une courte perte de contrôle et entraîner des conséquences catastrophiques.

Ces deux types de systèmes sont appelés systèmes temps réel *dur* pour les systèmes dont le total respect des échéances est indispensable, et systèmes temps réel *mou* lorsque les systèmes acceptent des ratées non contraignantes pour l'obtention du résultat final. Un type hybride, autorisant quelques fautes mais dont le fonctionnement général doit être considéré comme sous une contrainte de temps réel *dur* est appelée temps réel *ferme*. Le premier type concerne le fonctionnement de systèmes critiques, le second concerne les systèmes fournissant une qualité de service minimale et le dernier concerne les systèmes tolérant aux fautes.

La vérification qualitative telle que nous l'avons présentée dans le chapitre 2 est bien adaptée à la vérification des systèmes temps réel critiques, donc sous contrainte de temps réel *dur*. Ce sont donc ces systèmes que nous souhaitons modéliser en priorité.

4.1.4 Caractérisation du domaine

Parce que les systèmes que nous souhaitons vérifier sont des systèmes logiciels, nous nous limitons à l'expression de systèmes de *tâches* temps réel. Les tâches possèdent traditionnellement et comme nous allons le voir, des caractéristiques communes. Ces caractéristiques sont souvent assez générales pour être applicables à d'autres domaines sans grand changements : dans le domaine de la production industrielle, un utilisateur peut vouloir définir une tâche par un automate robotisé, mais l'action entreprise par le robot n'en garde pas moins un aspect très similaire à la tâche telle que nous la concevons dans un système logiciel ; planifier des cours ou des séminaires, ordonnancer l'envoi de messages sur un réseau, toutes ces actions possèdent également des caractéristiques communes et les cours, séminaires ou messages peuvent souvent être considérés comme des tâches à traiter. Nous ne nous intéresserons cependant pas aux possibles implications et applications de la notion de tâche, mais il nous semble important de souligner qu'elle est suffisamment abstraite pour être utilisable dans des domaines variés et possiblement étranger à celui de l'informatique.

Les tâches que nous entendons sont les entités gérées par les systèmes d'exploitation et plus particulièrement les systèmes d'exploitation temps réel. Une tâche sera donc le plus souvent un processus ou un *thread*.

Les types de systèmes de tâches temps réel qui nous intéressent expriment le comportement temporel du système, selon le point de vue de son flot de contrôle. La dynamique des données doit donc pouvoir être abstraite pour n'en retenir que les parties influençant le flot de contrôle. Les quantités temporelles sont également à définir, ce qui peut poser un véritable défi. Les programmes des systèmes temps réel sont quasiment systématiquement, à leur niveau le plus proche de la machine, dépourvu d'informations temporelles. La seule donnée temporelle est alors celle définie par le cumul des temps nécessaires aux instructions d'un programme. Ce problème peut être partiellement résolu par une analyse des pires temps d'exécution [WEE+08].

Mais cette problématique se pose surtout lorsque c'est le code qui sert de modèle. La méthode utilisée dans cette thèse pose à l'inverse un modèle censé représenter le système réel. Une implémentation du modèle est alors générée et cette implémentation doit être prouvée conforme. La problématique de la génération et de la conformité sont en dehors du champ d'étude de cette thèse.

Nous partons ainsi du principe que les temps du système sont connus, soit parce que c'est une définition, soit parce que d'autres analyses ont été effectuées pour fournir ces données.

Nous considérerons les systèmes de tâches basés sur la définition donnée par l'article de référence de Liu et Layland [LL73]. Le modèle de Liu et Layland définit une ressource processeur que les tâches doivent posséder pour s'exécuter. Une tâche admet une capacité (son temps d'exécution), une période et une échéance. Lorsqu'une tâche est périodique, son exécution est décomposée en instances réveillées (mise en attente de leurs exécutions) périodiquement. La période peut être assouplie pour rendre la

tâche sporadique [ABW95], c'est-à-dire que la période n'est plus fixe mais admet une borne minimale. Une tâche apériodique est une tâche sporadique dont la borne minimale est 0. Un offset peut décaler [Aud91] le (premier) réveil périodique d'une tâche.

Les tâches peuvent être préemptables et un système peut posséder plusieurs processeurs [RSL88].

Enfin, un système de tâches n'est rien sans un ordonnanceur. Le rôle d'un ordonnanceur est de décider l'ordre d'exécution des instances de tâche.

Le travail de thèse de J. Migge [Mig99] a apporté une formalisation d'un modèle de système de tâches qui est très proche de celui que nous souhaitons. Il a notamment intégré la possibilité de spécifier des fonctions de priorités utilisant les caractéristiques des tâches du systèmes. Ces fonctions sont ensuite utilisées pour donner le fonctionnement de l'ordonnanceur : la priorité d'une tâche est donnée par la fonction de priorité et l'ordonnanceur choisit la tâche possédant la priorité la plus élevée. Grâce à ce mécanisme, les politiques d'ordonnancement les plus usuelles sont spécifiables.

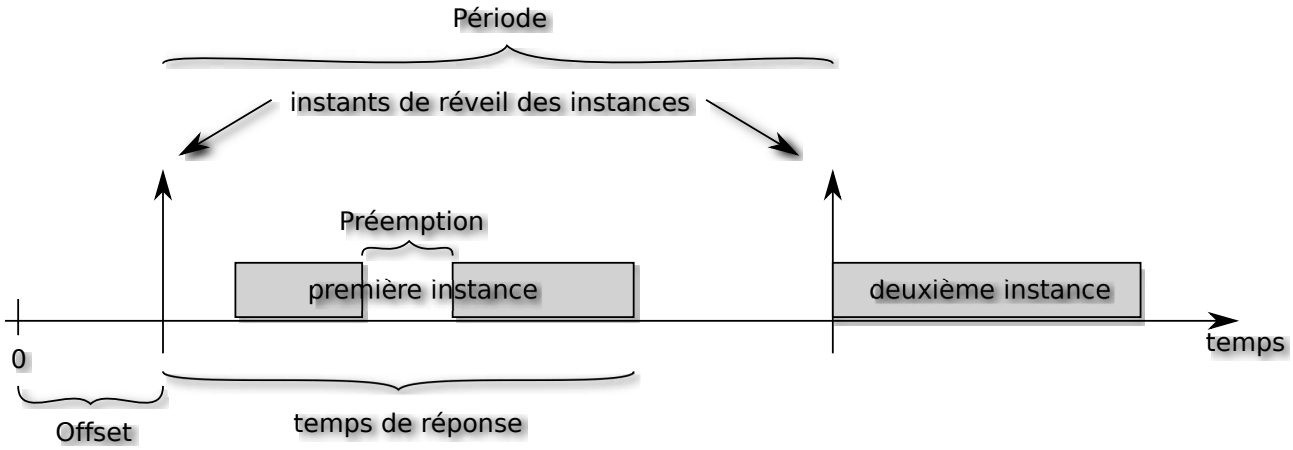


Fig. 4.3 – Les principales caractéristiques d'une tâche

Après cette brève introduction des systèmes de tâches que nous allons utiliser, nous souhaitons mettre en avant les points qui nous semblent essentiels et qui vont guider notre choix :

- le modèle de tâche doit être standard ;
- *toutes* les ressources sont modélisées au même titre que les tâches. Cela signifie qu'aucune ressource n'est implicite, ce qui implique que le (ou les) processeur(s) est une ressource comme les autres ;
- il doit y avoir une notion de préemption, au sens d'interruption/reprise de l'activité pour les tâches et d'appropriation/expropriation pour les ressources ;
- la possibilité de spécifier l'ordonnanceur et cela d'une manière la plus générique et simple.

4.1.5 des langages dédiés existants

Avant d'aller plus loin, nous souhaitons donner un rapide et partiel panorama de ce à quoi peuvent ressembler des langages dédiés proches de notre problématique. Nous avons choisi de présenter deux visions plutôt «industrielles» : Aadl et Marte. Langages à la fois concurrents et complémentaires, ils apportent une vision des besoins en termes généraux face à la problématique de la spécification des systèmes temps réel. Plus proches de notre problématique, nous donnerons un bref aperçu des automates de tâches ainsi que l'outil associé Times. Cet outil démontre la viabilité d'une approche de la vérification automatique à partir d'un langage dédié. Enfin nous donnerons une vision de ce que peut être un langage dédié à la conception de politique d'ordonnancement pour des systèmes d'exploitation temps réels, en ébauchant les possibilités du langage Bossa.

Aadl

Aadl [SAE04]est l'un des projets les plus avancés en ce qui concerne la modélisation des systèmes temps réel dans leur acception la plus générale. C'est un langage d'architecture, c'est-à-dire qu'il définit les liens entre les différents éléments du système. Le point de vue le plus caractéristique qui a fondé Aadl est que la connaissance d'un système temps réel passe par la prise en compte de tous les facteurs influençant l'évolution d'un système : le facteur logiciel, avec le flot de contrôle, le flot de données, mais également le facteur matériel sur lequel s'exécute l'application, la plate-forme d'exécution. La modélisation d'un système passe alors par l'intégration d'un ensemble d'éléments génériques primitifs, paramétrisables par des propriétés.

Les éléments dits «logiciels» peuvent être des *process*, donnant l'espace d'adressage d'une partie de l'application. Partageant l'espace d'adressage d'un *process*, on peut trouver des *groupes de threads* ou directement des *threads*. Ces derniers sont les éléments de plus bas niveau et sont les seuls éléments ordonnançables. Ce sont là des caractéristiques de type «code binaire» (code machine exécutable), mais on peut également donner une description textuelle, sous forme de code source, d'une partie du programme à l'aide de *subprogram* et de groupes de *subprogram*. Un *subprogram* même s'il est sous forme textuelle, est exécutable.

La partie matérielle est descriptible à l'aide de *processor*. C'est un élément à la fois matériel (ressource matérielle permettant d'exécuter le code) et logiciel (les fonctions du système permettant l'ordonnancement des *threads* font partie de *processor*) ; à l'aide de *virtual processors* permettant de décrire des machines virtuelles et des ordonnanceurs hiérarchiques ; de *memory* à laquelle les codes binaires et sources sont liés ; de *bus* permettant la communication et l'accès à la mémoire ; et les *devices*, décrivant les capteurs et actionneurs connectés au système par le biais du *bus*.

Ces éléments composent des *system*, qui peuvent changer de *mode* de fonctionnement si nécessaire.

Tous les éléments possèdent une interface, la partie *features*, et il est possible de déclarer des propriétés pour ces éléments à l'aide de *properties*. Le temps d'exécution, la période ou l'échéance d'un *thread* sont des exemples de *properties* (respectivement *Compute_Time*, *Dispatch_Protocol*, *Deadline*).

Si l'on rajoute à cela les annexes comportementales, permettant d'étendre le langage et les différents types de connexions permettant de relier les éléments entre eux, on se retrouve avec un langage permettant de spécifier une gamme vraiment impressionnante de systèmes temps réels.

Puisque nos buts sont si proches d'Aadl pourquoi ne pas l'utiliser comme langage de référence ? La principale raison est que l'étude du périmètre spécifiable par les *ipTPN* ne peut se faire *a priori*. En partant de Aadl, le risque est grand d'adopter son point de vue et ainsi de ne pas voir les manques du langage. De plus, si manque il y a, celui-ci doit être comblé à l'aide d'une annexe. Une telle annexe serait alors finalement assez proche de ce que nous voudrions avoir sans utiliser Aadl. Puisque Aadl possède la faculté de modéliser une gamme de systèmes très étendue, beaucoup de ses constructions deviennent inutiles dans une optique de vérification, ou trop puissantes. Ce qui rend le travail encore plus complexe : quoi garder ? quoi jeter ? pourquoi ?

Nous préférons définir le langage Pola de manière indépendante et si intégration il peut y avoir, cela doit être en soit un travail nécessitant une étude que nous ne pensons pas triviale et qui dépasse le cadre de cette thèse.

Il est à noter que l'outil Cheddar [SLNM05] permet la vérification par des méthodes analytiques, de systèmes Aadl. Les modèles supportés par Cheddar utilisent une annexe particulière à cet outil.

En ce qui concerne les outils liés à Aadl, on peut également citer la boîte à outils Topcased [FGC+06], dont le nom est l'acronyme en anglais de «boîte à outils *open source* pour la conception de systèmes embarqués critiques» et a pour objectifs de mettre en place les outils nécessaires à la conception de systèmes critiques. Cela peut aller de la simple modélisation, avec la panoplie d'éditeurs Aadl, Uml, etc. à la vérification en passant par une chaîne de transformation de modèles [BBF+08]. Cette thèse s'insère dans le cadre des objectifs de ce projet et les premiers prototypes de ce qu'allait devenir Pola ont été implémentés pour cette plate-forme.

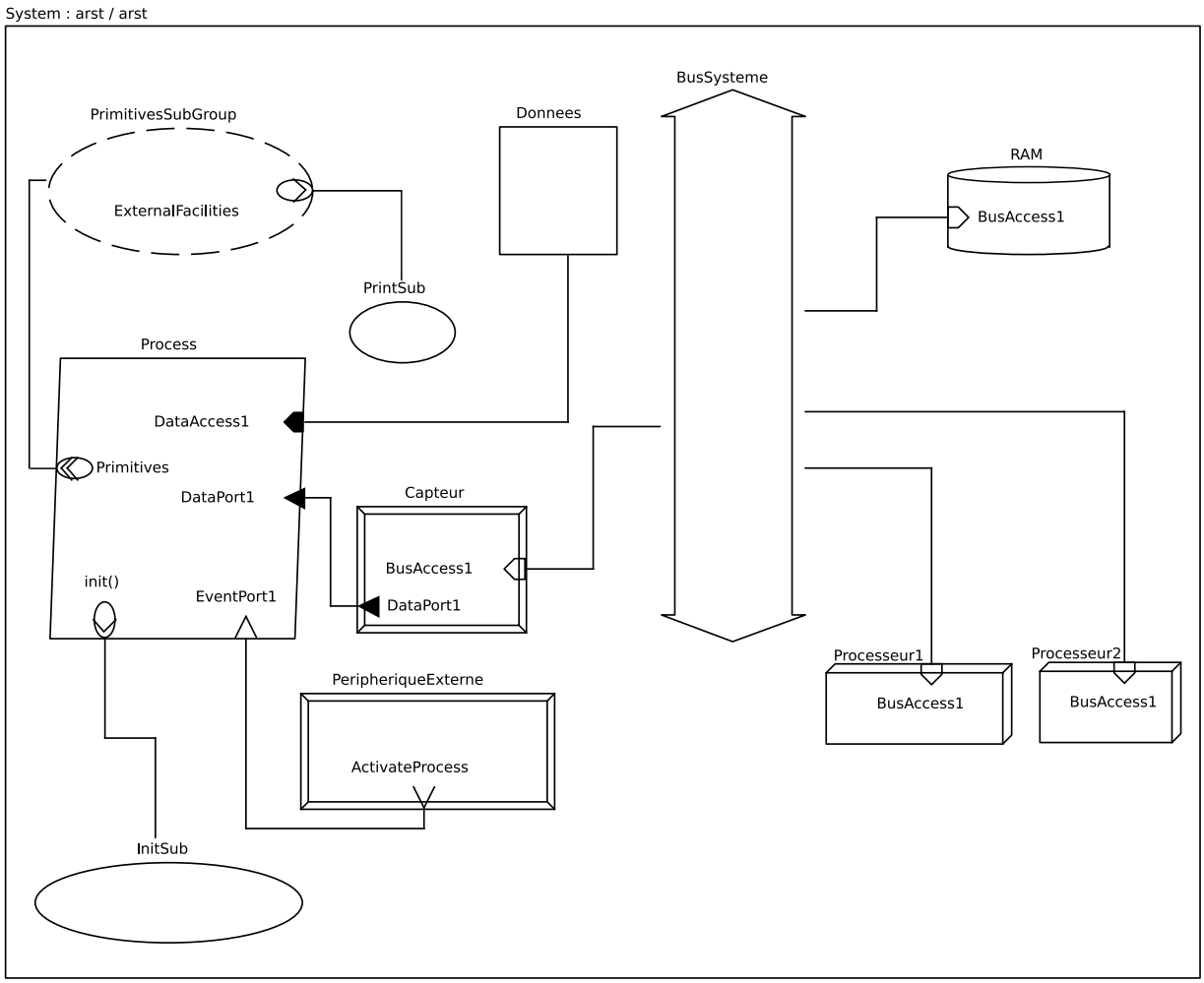


Fig. 4.4 – Un aperçu de la syntaxe graphique de Aadl

Marte

Marte [OMG08] est une approche plus récente que Aadl, en phase de finalisation et basée sur Uml [OMG09b][OMG09a]. Bien que concurrente, car reprenant beaucoup des buts et concepts de Aadl, il existe néanmoins des études permettant de jeter un pont entre les deux formalismes pouvant servir à une cohabitation des deux approches [FBDSG07][MAJ09].

Uml permet l'utilisation de *profils* pour étendre ses possibilités. Marte est un profil Uml comblant les manques en termes de spécification des systèmes temps réel d'Uml. Marte est la réponse à une requête de proposition issue de l'Omg, organisme normatif de la communauté modélisation objet.

Les possibilités de spécification de Marte sont nombreuses, à tel point que nous avons du mal à considérer la totalité du langage comme un DSL. Il est par contre des caractéristiques qui sont bien spécifiques aux systèmes temps réel, mais ceux-ci sont souvent noyés dans des concepts Uml qui, à notre avis, rendent le langage un peu moins accessible que peut l'être Aadl.

Les principales caractéristiques du profil Marte sont :

- de fournir un moyen de décrire des *propriétés non fonctionnelles*, c'est-à-dire des contraintes non inhérentes à la fonctionnalité du composant spécifié. La température d'un composant, sa période, *et caetera* sont des propriétés non fonctionnelles. Les contraintes sont exprimables à l'aide du langage de contrainte Vsl ;
- de fournir un modèle de temps très complet, sortant complètement du cadre de la problématique des langages dédiés. Le temps dans Marte peut être causal ou basé sur des horloges logiques ou physiques. La nature du temps utilisé peut être dense ou discret, mais toute les notions

temporelles définie le sont par des ensembles dénombrables d'instants. Les horloges définissent des instants dits de jonctions. Les instants de jonctions de différentes horloges peuvent alors être mis en relation pour soit définir l'évolution d'une horloge par rapport à une autre, soit contraindre l'évolution d'une horloge par rapport à une autre. A l'opposé des horloges logiques, on trouve les horloges chronométriques qui sont des horloges potentiellement basées sur des domaines denses, opérant un échantillonnage (propriété *resolution*) de ce domaine avec une précision plus ou moins grande (propriété *stability*) et pouvant aller à une vitesse donnée (par *skew*) et pouvant même avoir une accélération (donnée par *drift*). Pour plus de précision sur le modèle de temps défini dans Marte, voir [MA08] ;

- de fournir des primitives de ressources comme les ressources de stockage, de communication, de calcul, de périphérique, etc. Ces ressources (vues comme des services) sont allouées à leur clients par le biais d'un agent de change (*broker*) selon une politique d'accès. Un contrôleur de ressources peut aussi être défini permettant la création, destruction, etc. de ressources nécessaires, plus utilisés, etc. par les clients. La ressource *synch* permet de définir des synchronisations utilisant des protocoles d'exclusion mutuelle ;
- de permettre l'expression des liens logiciels/matériels ou simplement hiérarchiques logiciels/logiciels, matériel/matériels. Par exemple : la partie exécutoire d'un processus est liée à l'utilisation du processeur, ou bien la partie mémoire d'un processus utilise le mapping virtuel défini dans l'Os, qui a son tour est stocké sur le disque dur. Les allocations sont utilisables pour définir ce genre de relations ;
- une partie plus proche des fonctionnalités Aadl, donnant la possibilité d'utiliser des flots atomiques ou non, de définir des unités temps réel concurrentes (soit actives, soit passives) et de définir le matériel de manière précise.

Du point de vue de l'outillage pour la vérification, bien que le langage ne soit pas complètement finalisé, il existe déjà des méthodes comme l'utilisation d'Esterel pour effectuer la vérification de contraintes CCSL [AM09] et TimeSquare [FAMdS09] un outil de simulation de contrainte CCSL appartenant à la plate-formeOpenEmbeDD.

Même si ce n'est pas explicitement Marte qui est pris en compte, nous noterons ici l'existence de l'outil Mast [PHD01] permettant la vérification de systèmes par des méthodes analytiques, et possédant une syntaxe graphique par le biais d'un méta-modèle Uml.

Les automates de tâches

Un automate de tâche [FKPY07] est un automate temporisé avec invariants étendu par une notion de tâche : à toute location de l'automate on peut associer une tâche qui sera réveillée à chaque fois que le pointeur d'exécution entre dans la location. Une tâche est donnée par le triplet $T = (a, b, c)$, où a est le temps d'exécution minimal de T, b est le temps d'exécution maximal de T et c est le temps de réponse maximal que la tâche T doit respecter. Lorsqu'une tâche se termine, elle peut remettre une horloge à zéro.

Un automate de tâche est ordonnancé par une politique définie de manière externe. Le modèle suppose en effet une fonction d'ordonnancement des tâches classant celles-ci selon la priorité définie par la politique. La politique peut être préemptive ou non et présuppose la présence d'un seul et unique processeur à partager entre les tâches.

L'outil Times [AFM+02] implémente l'analyse des automates de tâches. En fait d'automates de tâches, cet outil utilise un modèle encore étendu puisque les tâches supportent un caractère périodique, sporadique ou contrôlé et possèdent une priorité, indispensable pour utiliser une politique arbitraire (aussi dite politique *offline*, car l'ordre d'exécution des tâches est défini à la spécification). On peut aussi signaler la présence d'un *offset*, décalage du premier réveil de la tâche et finalement un nombre de réveil maximal de la tâche. De plus, comme il est expliqué dans [FKPY07], si les temps d'exécution minimaux et maximaux d'une tâche sont égaux, alors la décidabilité de l'ordonnançabilité du modèle des automates de tâches est décidable, car il n'est pas besoin d'utiliser des chronomètres. L'outil Times tient compte de cela en ne permettant pas de spécifier d'incertitude sur la durée d'exécution,

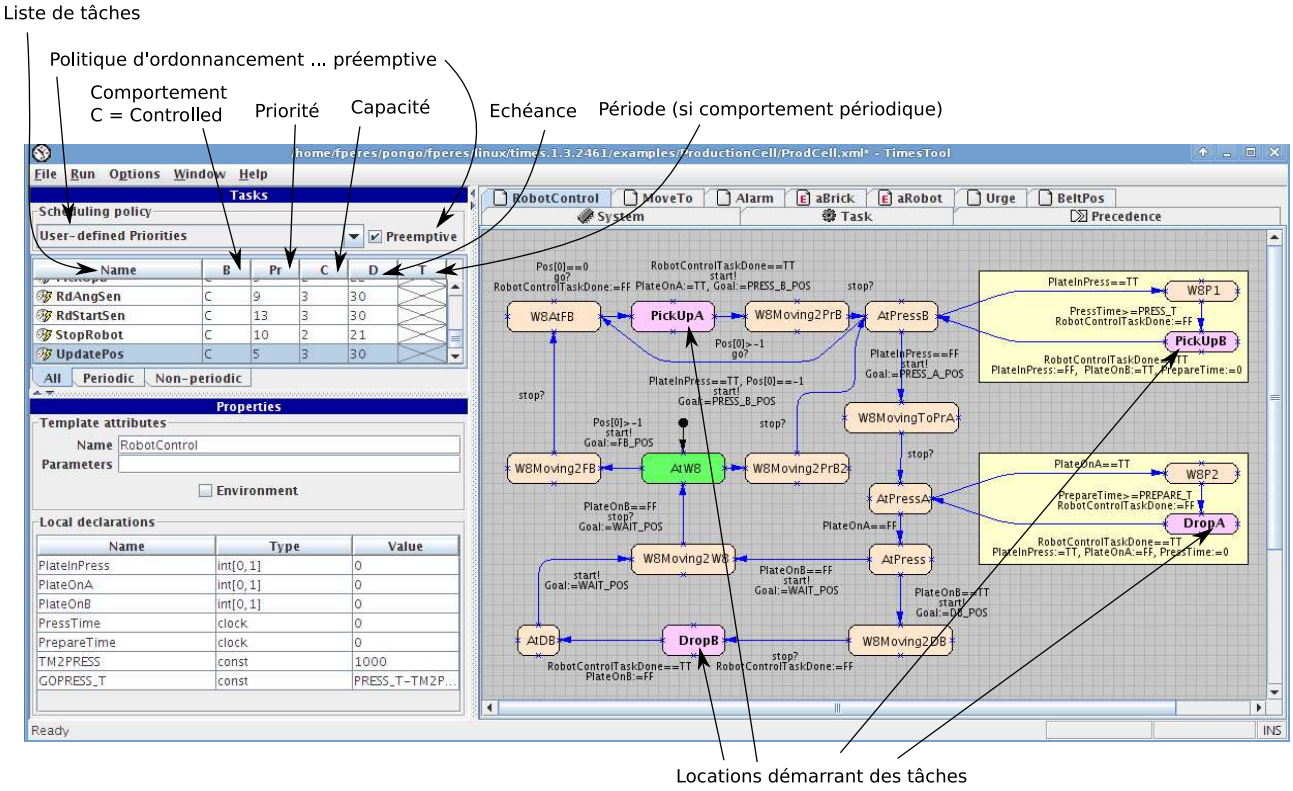


FIG. 4.5 – L'outil TIMES utilise des automates de tâches

permettant ainsi une analyse plus efficace et terminant à tous les coups.

Il faut également souligner la possibilité de décrire des précédences autrement que par la spécification d'un automate, par l'emploi d'un graphe de précédence supportant les opérations ET et OU ; la possibilité de décrire le comportement d'une tâche à l'aide d'un code C et au plus haut niveau de déclarer des *process* qui sont des automates de tâches paramétrisables permettant de contrôler les tâches dites contrôlables.

Le coté négatif de ces extensions est qu'elles ne sont pas réellement intégrées dans le formalisme des automates de tâches. C'est l'interface graphique qui segmente la spécification des différentes partie du modèle en autant de vues que d'extensions (précédence, paramètres des tâches, code C du comportement, etc).

Il nous semble également difficile de considérer du code C comme un moyen de spécification, car si les méthodes de vérification ont été explorées c'est bien pour vérifier des systèmes dont le code (souvent en C) est considéré comme trop complexe. Ceci vaut également pour la politique d'ordonnancement qui est une boîte noire. La seule garantie d'avoir effectivement un ordonnancement sous EDF est la confiance accordée en les développeurs de la dite politique. Confiance correspondant assez peu aux impératifs de la vérification formelle.

Néanmoins cet outil reste une référence en terme de vérification et démontre ce que l'on peut obtenir par des méthodes automatiques comme le *model checking*.

Bossa

BOSSA est un langage permettant la définition d'ordonnanceurs pour le noyau UNIX. Ecrire un ordonnanceur sans l'utilisation de cet outil, signifie que tout doit être codé «à la main», à partir des primitives fournies par le noyau UNIX. La programmation système UNIX est peu accessible et réservée à une «élite» (à tel point que *unix guru* est devenu un terme consacré pour désigner un expert UNIX). C'est dans une optique de simplification de l'accès aux primitives du noyau que le langage dédié à la spécification d'ordonnanceurs BOSSA a été conçu.

Un ordonnanceur est la donnée d'une politique et des ressources à allouer. Dans le cas de BOSSA, le

processeur étant l'unique ressource nécessaire à l'exécution d'un processus Unix, seule la description de la politique est nécessaire. Une politique est définie en Bossa de manière évènementielle : suivant ce qu'il se passe au niveau du système, le langage permet de changer l'état des processus. Les évènements sont par exemple l'arrivée ou la fin d'un processus, le blocage d'un processus, l'échéance d'un *timer*, etc. A l'aide de ces évènements, l'utilisateur peut décrire la réaction de l'ordonnanceur à ces évènements par le biais d'un langage dont la syntaxe est proche du langage C.

```
...
states = {
    RUNNING running : process;
    READY ready : select fifo queue;
    READY expired : queue;
    READY yield : process;
    BLOCKED blocked : queue;
    TERMINATED terminated;
}
...
```

```
...
On unblock.preemptive {
   if (e.target in blocked) {
   // check if the unblocking process
   // is more important than running
   if ((!empty(running)) &&
       (e.target > running))
   {
      running => ready;
   }
   e.target => ready;
}
...
```

Fig. 4.6 – Deux bouts de code Bossa montrant la définition des états et son style événementiel

4.2 Description du langage

Pour décrire un système temps réel, nous avons choisi de décomposer son modèle en plusieurs systèmes de tâches, composés de ressources ainsi que de tâches dont l'exécution est ordonnée selon des politiques d'ordonnancement. La description de l'ordonnanceur est découpée en deux parties : l'une décrivant la politique et l'autre l'allocation de ressources.

Systèmes. Un système (de tâches) est l'élément de plus haut niveau d'une spécification Pola. Celui-ci permet une meilleure décomposition du système global en parties fonctionnelles. A l'intérieur de ces parties, les noms des différents éléments sont protégés, cela en utilisant la hiérarchie des éléments pour leur attribuer un nom complet global. Ainsi, deux tâches pourront porter le même nom du moment qu'elles appartiennent à un système différent, le nom du système auquel elle appartiennent faisant office, en quelque sorte, de nom de famille.

Un système peut être *préemptable* (ajout du mot-clé **preemptable**) : toutes les horloges du système sont suspendues lorsque le système est désactivé. Par défaut, un système est considéré non préemptable, ce qui implique que toutes les horloges d'un tel système sont remises à zéro lorsque le système est désactivé.

Un système est composé de tâches, de ressources, de politiques d'ordonnancement, de définitions d'allocation de ressources et, le cas échéant, de comportements spécifiques.

Tâches. Les tâches sont les éléments opérateurs manipulables par l'ordonnanceur (les autres éléments opérateurs sont définis dans la partie comportement et peuvent échapper au contrôle de l'ordonnanceur). Elles peuvent être composées de plusieurs **actions**, auquel cas il est quasiment tout le temps nécessaire d'y adjoindre une glu comportementale entre les actions, permettant de déterminer le séquencement de ces dernières. Lorsque les actions sont parallèles, il n'y a pas besoin de cette glu, puisque dans ce cas il n'y a pas de séquencement à déterminer. Lorsqu'une seule action est définie, alors cette action représente complètement la *capacité* opérationnelle de la tâche. L'action est une sorte de boîte noire sans autre sémantique que celle de laisser s'écouler le temps. Plus y a d'actions, plus le comportement de la tâche est précis, sans toutefois aller plus loin dans la concrétisation que le séquencement des actions et leurs durées. En effet, plus il y a d'actions, plus ces actions vont représenter

de choses, apportant ainsi plus de précision, mais cela uniquement du point de vue temporel : on n'aura toujours pas plus d'informations sur la fonction de ces actions.

Une tâche admet également un intervalle pour définir sa périodicité. Si cet intervalle est réduit à une seule valeur alors la tâche est périodique, un intervalle non borné à droite défini une tâche sporadique, etc. Une tâche temps réel admet également une échéance (**deadline** en anglais) définissant l'instant après lequel la tâche (et donc tout le système) est considérée comme fautive. Le fait de rater une échéance est capturé par des évènements qui sont manipulables à la phase de vérification. Enfin, une tâche peut être déclarée préemptable (ou non), ce qui signifiera à l'ordonnanceur sa capacité (ou non) à l'interrompre.

Le premier réveil d'une tâche peut être spécifié en utilisant un intervalle *offset* : une fois le premier effectué, le générateur de réveil périodique prend le relais.

Une tâche doit également spécifier une *politique d'ordonnancement* selon laquelle la tâche sera ordonnancée. Cette politique est décrite dans une section spéciale, externe à la déclaration de la tâche. Afin de donner le choix le plus complet à l'utilisateur de spécifier la politique qu'il souhaite, il est possible de donner un niveau de priorité fixé en le spécifiant à l'aide du mot-clé **level**, niveau qui pourra ensuite être utilisé pour définir la politique.

Politique d'ordonnancement. Elle définit l'ordre de priorité des tâches. L'ensemble des tâches est partitionné selon les politiques mises en jeu. L'ordre (qui peut être partiel) ainsi défini est utilisé pour décider de la prochaine tâche à exécuter. Pour décrire des politiques, nous avons adapté le travail de [Mig99], dans lequel l'auteur a utilisé une représentation mathématique des tâches dont il se sert pour spécifier une large gamme de politiques. Il utilise ensuite cette spécification formelle pour en extraire des informations (les pires temps d'exécution notamment). Le résultat de cette adaptation est la possibilité de spécifier une politique à l'aide des deux opérateurs de tri **max** et **min** appliqués à une combinaison linéaire des caractéristiques des tâches de notre modèle. Pour construire la combinaison linéaire, on associe à chaque caractéristique une lettre qui sera le terme utilisé dans l'expression.

Ainsi, on associe la lettre majuscule **C** à la capacité de la tâche, qui est fixée arbitrairement à la somme des temps d'exécution des actions lorsqu'une tâche en comprend plusieurs. Si les actions ne sont pas enchaînées l'une à la suite de l'autre de manière connue *a priori* alors la spécification devient incohérente (un ordonnanceur capable de déterminer à la volée le pire temps d'exécution selon l'état du système est peu crédible et en dehors de notre champ d'étude). La période est associée au terme **P**, l'échéance à **D** et le niveau de priorité arbitraire à **L**. En appliquant un ordre (**min** ou **max**) à une combinaison de ces termes, on peut déclarer une politique, par exemple :

$$\textbf{\textit{policy}}\ \textit{RateMonotonic}\ \textbf{\textit{is}}\ \textit{min}\ P \tag{4.1}$$

Les termes précédemment définis correspondent à des caractéristiques statiques. Pour définir des politiques dynamiques, nous devons utiliser des caractéristiques dynamiques. Ainsi nous définissons **c** comme le temps cumulé d'exécution courant, par **p** ou **d** le temps courant écoulé depuis le dernier réveil. La différence entre **p** et **d** tient à leur intervalle de validité : **p** étant valide jusqu'au prochain réveil de la tâche et **d** jusqu'à l'échéance. Dans tous les cas, ces termes ne peuvent être définis que tant que la tâche est réveillée. Lorsque la tâche ne viole pas sa contrainte temps réel (i.e. son échéance), le domaine de validité de **d** se termine au même instant que celui de **c**. Les grandeurs utilisées pour définir une politique sont illustrées par la figure 4.7.

Voici quelques exemples de politiques d'ordonnancement courantes en Pola :

- Deadline monotonic $\Longleftrightarrow$ **policy** DeadlineMonotonic **is** min D
- Rate monotonic $\Longleftrightarrow$ **policy** RateMonotonic **is** min P
- User Defined $\Longleftrightarrow$ **policy** UserDefined **is** min L. L est défini par l'utilisateur en ajoutant à la déclaration de la tâche le mot-clé **level** suivi d'un entier naturel donnant la valeur du niveau de priorité.
- Earliest Deadline First $\Longleftrightarrow$ **policy** EarliestDeadlineFirst **is** min (D - d). Cette politique n'est pas prise en compte pour l'instant par l'outillage Pola. La différence $D - d$ est la valeur dynamique donnant le temps restant avant l'échéance. Déterminer la tâche possédant la plus petite de ces valeurs est donc équivalent à l'action d'une politique EDF.

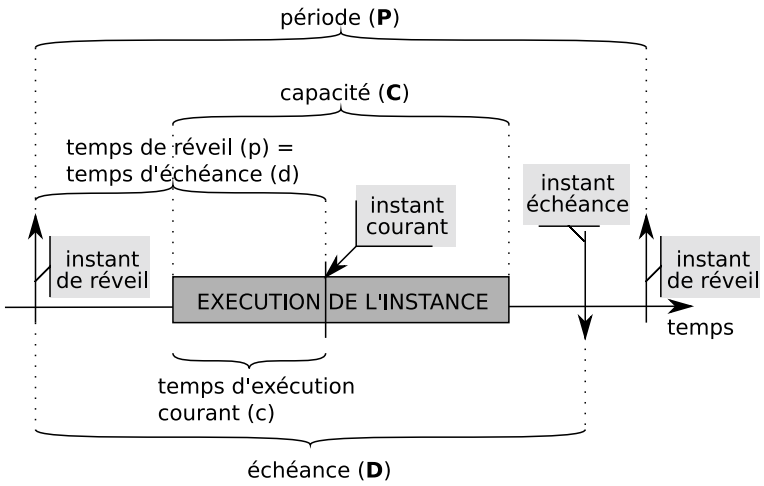


Fig. 4.7 – Grandeurs utilisables dans la définition d'une politique

– First come first served (a.k.a FIFO) $\Longleftrightarrow$ **policy** FIFO **is** max p. La tâche possédant la valeur pour **p** la plus grande signifie également que cette tâche est celle qui est réveillée depuis le plus longtemps. C'est donc elle qui est la première arrivée (et ce sera elle qui sera la plus prioritaire et donc la première servie).

Dans le chapitre suivant nous verrons la sémantique du langage Pola, donnée sous forme d'*ipTPN*. Nous savons que des politiques comme EDF ou FIFO, ne nécessitant qu'une différence d'horloges sont traduisibles à l'aide des *ipTPN*, car il est possible, avec un automate temporisé, de traduire un automate temporisé possédant des contraintes diagonales (contraintes utilisant des différences d'horloges) [BDGP98]. Mais ce n'est pas évident pour le cas général (pour n'importe quelle combinaison linéaire comprenant des termes «dynamiques») et donner la sémantique sous forme d'*ipTPN* de ces politiques dynamiques est laissé pour une étude ultérieure. En complément, nous pensons que l'exigence d'expressivité de la traduction des politiques dynamiques est si grande que c'est un réel défi pour beaucoup des modèles à temps dense actuels. Nous pensons que les automates hybrides pourraient les gérer, mais ce formalisme n'est pas vraiment applicable au *model checking* (à moins d'utiliser une très forte abstraction).

Allocations. Les politiques d'ordonnancement définissent l'ordre d'exécution des tâches. Elles ne sont pas suffisantes pour spécifier complètement un ordonnanceur qui se doit de connaître les ressources dont une tâche a besoin pour son exécution. Le rôle de l'élément allocation est donc de compléter l'information donnée par la politique d'ordonnancement en permettant de définir la liste de ressources utilisées par une liste de tâches. L'ordonnanceur aura alors pour rôle de donner toutes les ressources recensées par l'allocation à la tâche la plus prioritaire et cela de manière atomique (c'est-à-dire en un seul pas du système), dans la limite des possibilités d'attribution des ressources : si la tâche la plus prioritaire a besoin de plusieurs ressources, que l'une d'elles est utilisée par une autre tâche et qu'il n'est pas possible de retirer cette ressource de cette autre tâche moins prioritaire (soit que la ressource soit marquée non préemptable, soit que la tâche soit non préemptable), alors l'ordonnanceur ne donnera pas la main à la tâche la plus prioritaire et va examiner si c'est également le cas pour les tâches moins prioritaires.

Une ressource peut être attribuée à une tâche à la condition que cette ressource soit libre, c'est-à-dire qu'aucune tâche ne l'utilise ; ou bien lorsque la ressource n'est pas libre mais est préemptable et est possédée par une tache préemptable de plus faible priorité.

L'élément allocation permet d'attribuer un ensemble de ressources à une *tâche*. Comme les tâches sont subdivisibles en plusieurs actions et les actions nécessitant potentiellement des ensembles de ressources différents, alors les allocations sont liés aux actions. Ceci est réellement utile lorsque plusieurs actions utilisent des ensembles de ressources différents. Dans un tel cas, plusieurs allocations sont nécessaires qui seront désactivées à tour de rôle selon les actions activées par le comportement spécifique que l'utilisateur a utilisé pour enchaîner les actions. Ceci permet donc de calquer la dyna-

micité de l'allocation des ressources sur la dynamicité de l'exécution des actions au sein d'une tâche.

Le fait que la tâche reste le seul élément connu par l'ordonnanceur, au lieu des actions, entraîne une combinatoire bien moins importante que dans le cas où ce sont les actions qui sont le grain le plus fin visible par l'ordonnanceur. Ce gain contre l'explosion du nombre de cas que doit gérer l'ordonnanceur est obtenu au détriment de la concision du modèle (on doit coder dans le modèle ce qui aurait pu être incorporé primitivement dans l'ordonnanceur en utilisant les actions comme le niveau de précision le plus bas compris par l'ordonnanceur).

Comportements. Cette partie permet à l'utilisateur de définir un comportement spécifique (en utilisant le formalisme des réseaux de Petri) intégré à la spécification Pola. Cette intégration est rendue possible à l'aide de ce que nous nommerons les *accesseurs*. Ces accesseurs sont des étiquettes utilisables dans la partie définissant les comportements spécifiques : une opération de composition par les étiquettes, comme définie précédemment, lie alors le réseau de Petri donnant le comportement spécifique avec la sémantique définie également à l'aide des réseaux de Petri.

La présence des réseaux de Petri au sein du langage, en tant que langage de comportement est clairement une atteinte au principe de langage dédié. Sans vouloir la justifier, nous atténuerons l'importance de son influence en tant que langage non dédié en faisant trois remarques.

La première est que le recours aux réseaux de Petri dans un modèle indique une faiblesse du langage Pola, car s'ils sont utilisés c'est bien parce que les constructions de Pola ne permettent pas de décrire ce qu'il a été possible de spécifier à l'aide des réseaux de Petri. Une faiblesse de Pola peut soit être la marque d'un manque réel du langage en ce qui concerne la modélisation d'un pan du domaine, soit provenir d'une particularité au sein du modèle qui n'est pas spécifique au domaine des systèmes temps réel. Ce dernier cas dénote la tentative de modélisation d'un système temps réel «hybride», c'est-à-dire possédant une composante échappant aux problématiques temps réel. A moins de ne modéliser que des systèmes temps réel «purs», ce dernier cas sera souvent exhibé, c'est pourquoi nous ne pensons pas que le retrait pur et simple du langage de comportement soit un jour possible ni souhaitable.

La deuxième remarque que nous souhaitons faire est que l'utilisation des réseaux de Petri en tant que langage de comportement est souvent simple et se résume fréquemment à l'expression de la précédence entre deux actions ou deux tâches, ou bien l'activation/désactivation d'un certain élément du modèle. C'est-à-dire que c'est avant tout la caractéristique réactive du langage qui est utilisée, souvent sans informations temporelles, ces dernières étant la plupart du temps définies dans la partie déclarative du langage.

Finalement, la conception du lien entre le langage Pola et son langage de comportement a été pensée de telle façon qu'il soit possible de remplacer ce langage. Cette potentialité de remplacement vient de l'idée que du moment qu'un langage admet un étiquetage de ses éléments d'états et d'actions, il doit être possible (avec un minimum d'efforts) de l'utiliser comme alternative aux réseaux de Petri. Cette potentialité de remplacement du langage de comportement n'a jamais été mise à l'épreuve mais il semble indispensable que des propriétés préalables comme celles de composition soient à assurer avant de tenter une quelconque intégration un tant soit peu solide. Ce préalable mené à bien, nous pensons que cette modularité ne peut qu'être bénéfique et même que le remplacement *doit* être effectué de manière à coller au plus près à la philosophie ayant fait naître le langage : faciliter la spécification à l'utilisateur. Mais attention, ceci n'est pas fait pour enlever toute expertise à l'utilisateur, celui-ci devant de toute manière maîtriser son domaine. Faciliter la spécification signifie d'être au plus près du domaine de l'utilisateur, enlevant ainsi le plus possible le doute quant à la sémantique du modèle qui est spécifié. Nous pensons que, dans cet objectif, il n'est pas optimal d'utiliser un mélange bas niveau/ haut niveau d'abstraction, comme c'est le cas lorsqu'on utilise les réseaux de Petri comme langage de comportement.

La syntaxe du langage va maintenant être progressivement dévoilée sur des études de cas.

4.2.1 Groupes de tâches et points de réordonnancement à la Osek

Osek [OSE] est un standard définissant un système d'exploitation temps réel adapté à l'industrie

automobile. Nous en avons extrait deux mécanismes intéressants liés à la façon dont sont suspendues/exécutées les tâches. Nous allons tout d'abord montrer les points de réordonnancement, suivis des groupes de tâches et enfin de la façon dont ils sont tous deux codés en Pola.

Points de réordonnancement

Prenons ce qui est dit dans le standard :

> Dans le cas d'une tâche non préemptable, le réordonnancement aura lieu dans les cas suivants :
> 1) la tâche s'est terminée avec succès ;
> 2) la tâche s'est terminée avec succès et active une tâche la succédant ;
> 3) appel explicite de l'ordonnanceur ;
> 4) une transition menant dans l'état *attente*

Pour bien comprendre ce que tout cela signifie, il nous faut également le graphe montrant les changements d'états d'une tâche (figure 4.8).

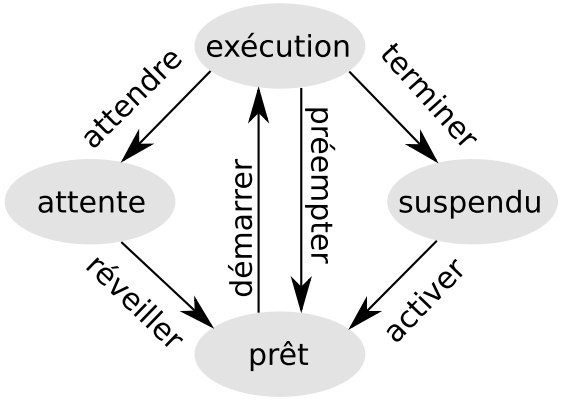


Fig. 4.8 – Les différents états d'une tâche Osek

Lorsqu'une tâche est initialisée mais pas encore activée ou lorsqu'elle se termine de manière normale (avec succès), elle se trouve dans l'état *suspendu*. Dans le cas où la terminaison n'a pas lieu de manière correcte le système reprend la main (par une interruption par exemple). Ce cas-là sort de notre cadre d'étude, car des comportements spécifiques, qui peuvent être très complexes, sont à mettre en place.

Une tâche Osek est dans l'état *exécution* lorsque toutes les conditions sont remplies et que l'ordonnanceur lui a attribué le processeur. L'état *attente* est atteint par la tâche sur un appel service du système la suspendant. La tâche ne peut être ordonnancée tant qu'elle est dans cet état. L'état *prêt* est le plus courant : la tâche est prête à être exécutée mais n'a pas encore reçu les ressources nécessaires (dans notre cas, le processeur).

De notre point de vue, préciser que la tâche peut en activer une autre avant le déclenchement du réordonnancement ne nous intéresse pas. Ceci est important pour le standard car alors la fonction utilisée par la tâche n'est pas la même dans les deux cas.

La figure 4.9 montre comment on peut faire correspondre ces états à un équivalent Pola.

En Pola, la tâche n'a que deux états possibles : *released* (réveillée) ou *not released* (son contraire). Seule une tâche marquée *released* est prise en compte par l'ordonnanceur.

L'état Osek *exécution* correspond pour une tâche Pola à être marquée *released* et posséder toutes les ressources nécessaires à son exécution. En Pola, cet état peut être partagé par plusieurs tâches lorsque le système est multi-processeur. Or un système Osek n'en possède qu'un. C'est pourquoi il ne peut y avoir effectivement qu'une tâche à la fois dans l'état *exécution*. Une fois la tâche terminée, elle n'est plus marquée *released*, d'où la correspondance entre l'état *suspendu* et *not released*. Lorsque la tâche peut être préemptée (c'est-à-dire lorsqu'elle est préemptable ainsi qu'au moins l'une des ressources utilisées) l'interruption se fait en enlevant une ressource préemptable, ce qui correspond à l'état *prêt*.

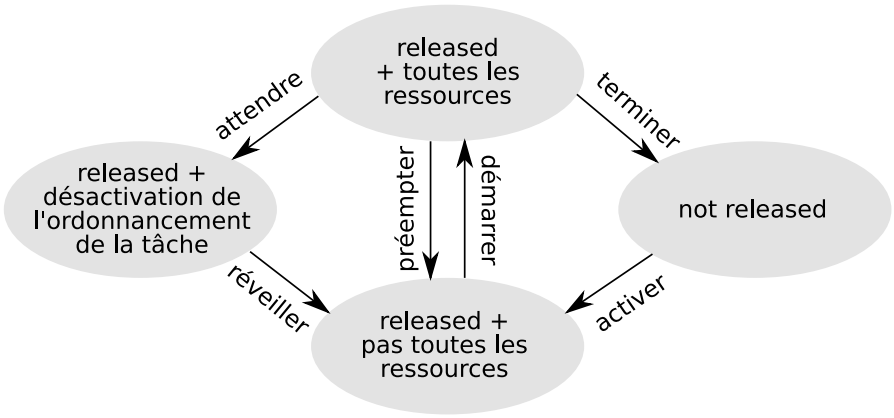


Fig. 4.9 – Les différents états d'une tâche Osek, interprétés en Pola

L'état d'attente est un peu plus délicat à faire correspondre en Pola. En effet, une tâche qui est réveillée est ordonnançable. Or l'état *Attente* du modèle Osek, nécessite que la tâche soit réveillée mais pas ordonnancée. Il n'est pas question de modifier son état «réveillée» puisqu'elle n'est pas finie : bien qu'en attente, la tâche est donc toujours réveillée. Puisqu'en état d'attente une tâche ne doit pas être ordonnançable, mais reste malgré tout réveillée, et puisque toute tâche réveillée est potentiellement ordonnançable, notre seule alternative est de désactiver momentanément l'ordonnanceur. Ceci ne peut être accompli que grâce à un ajout de comportement spécifique. D'après le standard Osek, la tâche ne peut aller dans cet état que par l'appel à un service. Une fois en attente, seul un signal externe peut la faire passer dans l'état *prêt*. Les ressources normalement occupées par elle sont libérées lors de l'appel au service. Le service en question, ou plutôt une abstraction de ce service, correspond donc au comportement spécifique que l'on devra donner en Pola pour spécifier le passage dans l'état *attente*.

Le dernier cas de réordonnancement possible est un cas simplifié du cas précédent, puisque dans celui-ci, la tâche décide de s'interrompre et de donner la main à l'ordonnanceur mais cette fois-ci, elle reste ordonnançable.

Nous allons maintenant présenter sur la figure 4.10, un exemple de modèle Pola pour lequel une tâche, t_1, passe par tous ces états.

```
system osek is
  res processeur is not preemptable
  task t1 is
    action a1 in [3,3] with alloc1 giveback
    action a2 in [3,3] with alloc1 giveback
    action a3 in [3,3] with alloc1 endoftask
    period [20,20]
    policy RM
    behavior is
      tr t1 a1 alloc -> a2
      tr t2 a2 -> a3
      tr t3 a3 -> a1
      pl a1 (1)
      lb osek.t1.a1 t1
      lb osek.t1.a2 t2
      lb osek.t1.a3 t3
      lb osek.alloc1.t1.active alloc
  end
  task t2 is
    action a1 in [2,2] with alloc1
    period [4,4]
    offset [2,2]
    policy RM
    behavior is
      tr t1 alloc ?-1 -> alloc
      tr t2 alloc ?1 ->
      lb osek.t2.a1 t1
      lb osek.t2.a1 t2
      lb osek.alloc1.t1.active alloc
  end
  allocation alloc1 is
    resources processeur
    tasks t1,t2
  policy RM is min P
end
```

Fig. 4.10 – Un modèle Pola présentant tous les points de réordonnancement Osek

Le système donnant les points de réordonnancement est appelé *osek*, il est déclaré par la structure :

system osek **is**
. . .
end

Le processeur est déclaré par la ligne :

res processeur **is not preemptable**

Puisque la ressource, *processeur*, est **not preemptable** toutes les tâches qui l'utilisent (comme c'est le cas de l'exemple 4.10) sont à leur tour non préemptables, même si ceci n'est pas explicitement indiqué. L'ordonnanceur n'a pas le droit de transférer une ressource d'une tâche à une autre plus prioritaire si cette ressource est **not preemptable**. Seule une ressource non préemptable libre, c'est-à-dire qu'aucune tâche ne possède, peut être allouée à la tâche la plus prioritaire par l'ordonnanceur (par le biais de l'allocation).

La déclaration de la tâche t_1 est contenue dans la structure :

task t1 **is**
. . .
end

Une action est donné par une déclaration similaire à celle qui suit :

action a1 **in** [3,3] **with** alloc1 **giveback**

Cette ligne donne le nom de l'action (a1), sa durée (3 unités de temps exactement) et l'allocation utilisée par l'action, indiquant les ressources nécessaires à son exécution. Ici, puisque l'allocation *alloc1* alloue le processeur, cela signifie que l'action *a1* a besoin du processeur pour s'exécuter. Le mot-clé **giveback** précise que lorsque l'action se termine, les ressources sont libérées sans pour autant terminer la tâche. Le mot-clé **endoftask** signale que l'action termine la tâche. Lorsqu'il n'y a qu'une seule action, celui-ci est sous-entendu. De même qu'il est sous-entendu qu'une action marquée **endoftask** restitue également les ressources.

La période et l'*offset* sont déclarés en précédant leur valeur (sous forme d'intervalle) des mots-clés **period** et **offset**.

Un comportement spécifique est déclaré sous les mots-clés **behavior is**. Un comportement peut être déclaré au sein d'une tâche ou au niveau système. Les noms des éléments définis dans un comportement sont protégés par une hiérarchie de nom : une transition t_1 définie dans le comportement d'une tâche T_1, au sein d'un système S, s'appellera $S.T_1.t_1$. En toute rigueur, le comportement n'est pas spécifiquement attaché à la tâche, et pourrait très bien être donné ailleurs. Mais dans un souci de clarté, il est préférable de mettre le comportement lié aux caractéristiques de la tâche dans la partie comportement de la tâche en question.

La tâche t_1 est découpée en trois actions dont la durée est exactement de trois unités de temps et utilisant la même allocation *alloc1* (ce qui n'est pas étonnant puisqu'il n'y a qu'une seule ressource). Les trois actions rendent les ressources dès qu'elles s'achèvent (mot-clé **giveback** ou **endoftask**). La troisième action termine la tâche (mot-clé **endoftask**).

Le comportement de la tâche *t1* est donné par une notation textuelle des réseaux, compatible avec celle utilisée par l'outil Tina [BRV03]. Une transition est déclarée par le mot-clé **tr** suivi du nom de la transition, de son intervalle de tir (on peut ne pas le mettre s'il est égal à $[0, \infty[$), enfin vient les listes de noms des places préconditions et postconditions, séparée par ->. Une place est déclarée par le mot-clé **pl** suivi de son nom et de son marquage initial. Les places n'ayant pas de jetons initialement n'ont pas besoin d'être déclarés par **pl**, sauf cas rares. L'instruction **lb** donne une étiquette (le premier argument), qui est le plus souvent un accesseur Pola, à un élément Petri (le deuxième argument). Le comportement de la tâche *t1* est donné sous forme graphique par la figure 4.11.

Ce comportement utilise quatre accesseurs : un d'état, *osek.alloc1.t1.active*, qui identifie la place *alloc* à l'activité de l'allocation *alloc1* de la tâche *t1* et trois d'actions *osek.t1.a1*, *osek.t1.a2*, *osek.t1.a3*, identifiant les trois transitions *t1*, *t2* et *t3* aux évènements de terminaison des actions *a1*, *a2* et a3.

Ce comportement est cyclique : les trois actions *a1*, *a2* et *a3* sont enchaînées l'une à la suite de l'autre. Lorsque la première action est terminée, la ressource *processeur* est libérée et la partie de

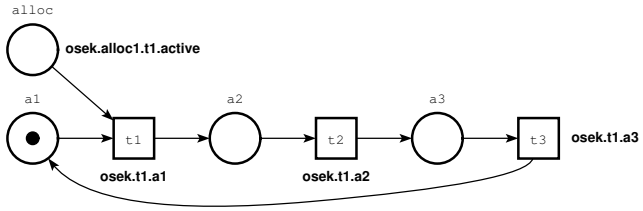


Fig. 4.11 – comportement de la tâche *t1*

allocation liée à *t1* est désactivée. En désactivant ainsi cette partie de l'allocation, l'ordonnanceur ne prend plus du tout en compte la tâche *t1*. Ainsi, lorsque la tâche *t1* a terminé sa première action elle se retrouve dans l'état *attente* du modèle de comportement des tâches Osek. La tâche *t2* doit alors réveiller la tâche *t1*, en réactivant l'allocation de *t1*, pour que l'action *a2* puisse avoir lieu.

La deuxième tâche, à cause de son *offset*, est réveillée pour la première fois deux unités de temps après le démarrage du système : la tâche *t1* est alors en train d'exécuter l'action *a1*. Elle est plus prioritaire mais ne peut pas prendre la main car la ressource *processeur* n'est pas préemptable. Une fois l'action *a1* terminée, la tâche *t1* se met en attente, permettant à la tâche *t2* de s'exécuter.

La tâche *t2* est constituée d'une seule action. Son comportement est plus simple que celui de *t1*. Deux nouveautés sont introduites : nous avons utilisé un *read arc* (noté **?** suivi du poids, et représenté graphiquement par un rond plein) et un arc inhibiteur (noté **?-** suivi du poids, et représenté graphiquement par un rond creux). Les deux transitions sont liées à la même action *osek.t2.a1*, c'est-à-dire l'unique action *a1* de la tâche *t2*. La place *alloc* est liée à la (ré)activation de l'allocation de *t1*. La figure 4.12 donne ce comportement sous forme graphique.

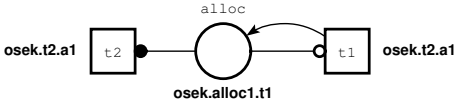


Fig. 4.12 – comportement de la tâche *t2*

Ce comportement signifie qu'à la fin de la tâche *t2*, celle-ci réactive l'allocation pour *t1*, mais seulement si c'est nécessaire, c'est-à-dire seulement si l'allocation est désactivé. D'où les deux transitions du comportement de *t2* modélisant ces deux cas possibles.

La tâche *t1* reprend son exécution sitôt la tâche *t2* terminée, jusqu'à ce que l'action *a2* soit terminée. A ce moment-là, la tâche t_1 demande explicitement un réordonnancement, sans toutefois se mettre dans l'état *attente*. La tâche t_2 a entre-temps été réveillée et puisqu'elle est prioritaire, prend donc la main. Son exécution se termine à l'instant où un nouvel évènement de réveil intervient ne laissant pas le moindre instant à la tâche *t1*, ne lui permettant ainsi pas de reprendre la main avant la fin de cette nouvelle instance de *t2*. La tâche *t1* se termine après l'exécution de l'action *a3*. Sa période correspond à 5 activations de la tâche *t2*, ce qui implique que son comportement après cette nouvelle activation est le même que celui à l'état initial. Ce comportement est repris pour plus de clarté sur le chronogramme 4.13.

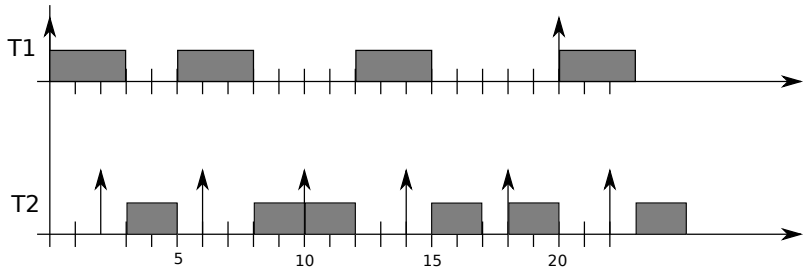


Fig. 4.13 – Chronogramme du comportement du modèle Pola précédent

La politique utilisée pour ordonnancer les tâches t_1 et t_2 est RM définie par la déclaration :

policy RM **is** min P

La caractéristique utilisée pour ordonnancer les tâches est la valeur statique de la période : plus la période de la tâche est petite, plus la tâche est prioritaire. C'est la politique couramment appelée *Rate Monotonic*.

Finalement, la déclaration de l'allocation stipule que les deux tâches *t1* et *t2* doivent posséder la ressource *processeur* pour s'exécuter :

allocation alloc1 **is**
resources processeur
tasks t1,t2

Groupes de tâches

Les groupes de tâches sont une caractéristique des systèmes Osek qui a retenu notre attention car ils permettent de combiner les aspects préemptifs et non préemptifs des tâches. La définition du standard est suffisamment courte pour être intégrée ici.

> «*Pour des tâches ayant une priorité plus faible ou égale à la priorité la plus haute dans un groupe, les tâches du groupe se comportent comme des tâches non préemptables : le réordonnancement a seulement lieu aux points de réordonnancement. Pour des tâches avec une priorité plus élevée que la priorité la plus élevée au sein d'un groupe, les tâches du groupe se comportent comme des tâches préemptables.*»

Formellement : soit un ensemble T de tâches, soit $P(t \in T) \in \mathbb{N}$ la fonction donnant la priorité de la tâche t et $t \rhd t'$ signifie que t' est préemptable par t. Si $G \subseteq T$ est un groupe de tâches, alors :

- $(\forall t \in T)(\forall t' \in G)(P(t) > P(t') \Rightarrow t \rhd t')$
- $(\forall t \in T)(\forall t' \in G)(P(t) \leqslant P(t') \Rightarrow t \not\rhd t')$

De la dernière proposition on en déduit une autre : $(\forall (t,t') \in G)(t \not\rhd t')$. En effet, toutes les tâches t d'un groupe satisfont $P(t) \leqslant P(t')$.

La politique d'ordonnancement agit de la façon suivante :

$$(\forall (t,t') \in T^2)(P(t) > P(t') \Leftrightarrow t \rhd t') \tag{4.2}$$

Le groupe de tâche implique que :

$$(\forall t \in T \setminus G)(\forall t' \in G)(P(t) > P(t') \Leftrightarrow t \rhd t') \tag{4.3}$$

Cela signifie que la notion de groupe est un nouveau filtre posé sur la politique d'ordonnancement sous-jacente, inhibant l'effet de cette politique au sein même du groupe. En dehors des groupes, la politique opère normalement.

Exemple La configuration présentée en figure 4.14 donne deux groupes comprenant chacun deux tâches. La relation d'ordre définie par la politique d'ordonnancement est représentée par le chemin fléché. Ainsi, la tâche *t1* est de plus faible priorité et *t4* est de plus forte priorité. Nous supposerons dans la suite que toutes les tâches sont préemptables. De manière "naturelle" la politique d'ordonnancement donne les relations suivantes : $t_2 \rhd t_1$, $t_3 \rhd t_2 t_1$; $t_4 \rhd t_3 t_2 t_1$. Le fait de rajouter des groupes va donc restreindre l'ensemble de ces relations. C'est le cas de

- $t_2 \rhd t_1$, car $P(t_2) < P(t_3) \wedge t_1 \in \mathit{Groupe}(t_3)$;
- $t_3 \rhd t_2$, car $P(t_3) < P(t_4) \wedge t_4 \in \mathit{Groupe}(t_2)$;
- $t_3 \rhd t_1$, car $t_3 \in \mathit{Groupe}(t_1)$;
- $t_4 \rhd t_2$, car $t_4 \in \mathit{Groupe}(t_2)$

où $\mathit{Groupe}(x)$ est l'ensemble des éléments appartenant au(x) même(s) groupe(s) que x.

Il ne reste donc plus que $t_4 \rhd t_3 t_1$.

Dans cet exemple, la tâche t_2 et t_4 ne seront préemptées par aucune tâche. Ce qui implique que l'on peut leur attribuer l'option *not preemptable*. Mais ce n'est pas le cas des tâches t_1 et t_3, qui ne

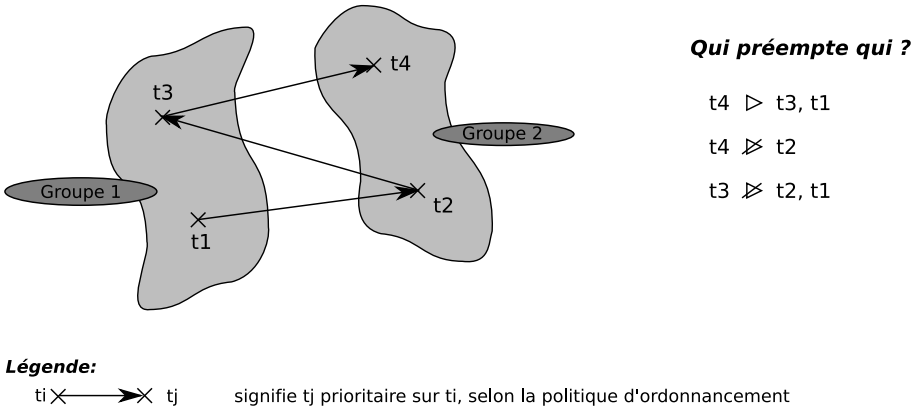


Fig. 4.14 – Un exemple de groupes de tâches

sont pas préemptables par t_1, t_2 ou t_3 mais le sont par t_4. Ceci implique donc que l'on ne peut pas modéliser ce comportement en n'utilisant que l'option *not preemptable* des tâches.

De même, il n'est pas possible de dire que le processeur, utilisé par toutes les tâches, est *not preemptable*, car cela ne permettrait pas à t_4 de préempter t_3 ou t_1.

Une solution possible est d'ajouter une autre ressource. Le processeur est obligatoirement *preemptable*, d'où l'ajout d'une deuxième ressource *non preemptable*.

res processeur **is preemptable**
res R1 **is not preemptable**

Dans cet exemple, nous avons déjà vu que la définition des groupes impose $t_4 \triangleright t_3 t_1$ et la politique sous-jacente $t_3 t_1 \not\triangleright t_4$. Si on déclare :

allocation alloc1 **is**
  resources processeur
  tasks t1,t3,t4

on respecte $t_4 \triangleright t_3 t_1$ et $t_3 t_1 \not\triangleright t_4$, mais on ne tient par compte du groupe 1, stipulant que $t_3 \not\triangleright t_1$, or selon la politique, $t_3 \triangleright t_1$ est possible. Il faut donc neutraliser l'effet de la politique. Nous allons pour cela nous servir de la deuxième ressource *R1* afin de régler ce problème.

allocation alloc1 **is**
  resources processeur
  tasks t4
allocation alloc2 **is**
  resources processeur, R1
  tasks t1,t3

Ainsi $t_3 \not\triangleright t_1$. Il reste à intégrer la tâche t_2 qui doit respecter $t_2 \not\triangleright t_3 t_1$ à cause de la sémantique des groupes de tâches et $t_2 \not\triangleright t_4$ à la fois à cause de la politique et à cause de son appartenance au groupe 2. Ceci correspond aux mêmes conditions que pour les tâches t_3 et t_1, nous pouvons donc regrouper les trois tâches dans *alloc2*.

Solution générale Si nous avons pu trouver une solution pour l'exemple ci-dessus, celle-ci ne s'en révèle pas moins décevante : nous n'en avons pas extrait de méthode généralisable à tous les types de systèmes et la notion de groupe est complètement cachée.

Au lieu de partir de l'analyse du système pour en tirer un ensemble de contraintes que l'on modélisera, nous allons repartir de la sémantique des groupes et essayer de la modéliser le plus directement possible.

De la sémantique des groupes, nous savons que toutes les tâches au sein d'un groupe ne peuvent se préempter entre elles. Ceci est aisément codé par l'utilisation d'une ressource non préemptable pour toutes les tâches au sein de ce groupe.

res groupe_x **is not preemptable**
...
allocation alloc_Groupe_x **is**
resources ..., groupe_x
tasks $t_i, \ldots, t_j$

Il reste donc deux cas à traiter :

(1) $(\exists t)(\forall t' \in G)(P(t) > P(t') \vee P(t) < P(t'))$. Une tâche t est soit plus prioritaire, soit moins prioritaire que toutes les tâches du groupe G. Le comportement de t dans ces cas là est exactement équivalent à celui dicté par la politique d'ordonnancement, il n'y a donc rien à faire de plus.

(2) $(\exists t)(\exists t' \in G)(\exists t'' \in G)(P(t') \leqslant P(t) < P(t''))$. Une tâche t est à la fois moins prioritaire et plus prioritaire que deux tâches appartenant au même groupe. Ce cas est à modéliser car la politique prévoit $t \triangleright t'$, or le groupe G impose $t \not\triangleright t'$.

Le deuxième cas peut être modélisé à l'aide de nouvelles ressources : une nouvelle par couple (t, e) où t est une tâche et $e = \{t' \in G \mid t' \leqslant t \wedge (\exists t'' \in G)(t < t'')\}$, l'ensemble des tâches du groupe G qui ne sont pas les plus prioritaires du groupes et qui sont moins prioritaires que t. Cette ressource est nécessaire à l'exécution de la tâche t et à celles qui sont dans e.

L'exemple précédent est alors modélisé de la manière suivante :

res processeur **is preemptable**
res G1 **is not preemptable**
res G2 **is not preemptable**
res t3_t2 **is not preemptable**
res t2_t1 **is not preemptable**

allocation alloc_t4 **is**
resources processeur, G2
tasks t4

allocation alloc_t3 **is**
resources processeur, G1, t3_t2
tasks t3

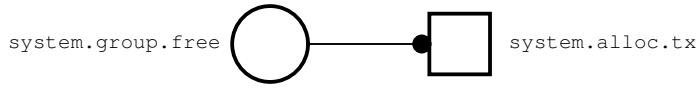
allocation alloc_t2 **is**
resources processeur, G2, t3_t2, t2_t1
tasks t2

allocation alloc_t1 **is**
resources processeur, G1, t2_t1
tasks t1

Cette solution est applicable de manière générale mais comporte un gros problème de passage à l'échelle : on est obligé d'énumérer tous les cas possibles et il peut y en avoir beaucoup. Elle apporte malgré tout une visibilité des groupes qui est appréciable.

Pour faire mieux, il faut utiliser d'autres mécanismes de Pola. En effet la sémantique des groupes requiert que l'ordonnanceur soit encore plus contraint que par la politique seule, c'est-à-dire que l'on doit empêcher certains comportements qui auraient été possibles si l'on n'avait eu que la politique d'ordonnancement comme contrainte d'ordonnancement. Ceci implique donc que certaines possibilités d'allocations de ressources doivent être empêchées.

Avoir ajouté une ressource liée au groupe, empêchant les tâches du même groupe de se préempter, va nous faciliter la tâche. Lorsqu'une telle ressource est libre, cela signifie que la voie est libre pour l'allocation, alors que dans le cas contraire, on doit empêcher l'allocation d'avoir lieu. Par conséquent, toute tâche t_x telle que $(\exists t' \in group)(\exists t'' \in group)(P(t') \leqslant P(t_x) < P(t''))$ va s'enrichir du comportement suivant :



La place est liée à l'accesseur permettant de connaître l'état actuel de la ressource *group*. Si elle est libre, un jeton est donc présent dans la place. L'arc est un *read arc* : la transition est sensibilisée lorsqu'un jeton est présent et son tir ne change pas le marquage de *system.group.free*. Lorsqu'une

tâche t' du groupe *group* s'exécute, elle utilise forcément la ressource et par conséquent la place *system.group.free* n'a aucun jeton, ce qui empêche de tirer la transition liée à l'allocation de t_x.

L'exemple de la figure 4.14, appliqué à la solution décrite ci-dessus, donne le code Pola suivant :

```
system groups is
  res processeur is preemptable
  res G1 is not preemptable
  res G2 is not preemptable

  allocation alloc1 is
    resources processeur, G2
    tasks t2, t4

  allocation alloc2 is
    resources processeur, G1
    tasks t1, t3
  end
```

```
task t2 is
  ...
  behavior is
    tr t0 res ?1 ->
    lb groups.alloc1.t2 t0
    lb groups.G1.free res
end
task t3 is
  ...
  behavior is
    tr t0 res ?1 ->
    lb groups.alloc2.t3 t0
    lb groups.G2.free res
end
```

Comme on peut le voir, cette solution est plus linéaire et requiert moins de ressources et de blocs allocations que la solution précédente.

Il est à noter que le langage Pola que nous étudions à présent est volontairement concis et épuré. Il est bien sûr préférable, plutôt que de coder les groupes à la main à chaque fois, d'intégrer ce mécanisme directement dans le langage. Ceci n'est actuellement pas fait, mais nous avons montré ici que cette intégration était possible, une fois la syntaxe trouvée. Par exemple, en ajoutant le champ "**group** x" dans les champs de tâche, où x est un entier naturel.

Il n'existe pas un seul niveau d'abstraction pour concevoir un D.S.L.. Pola est un langage dédié à la spécification des systèmes de tâches temps réel, même si l'ajout de **group** ne demanderait sûrement pas beaucoup d'efforts, il convient de se demander à quoi et à qui servirait cet élément. En l'occurrence cet élément ne sert que pour la spécification d'un système Osek. Cette spécificité amène une autre question : ne serait-il pas préférable de concevoir un langage dédié propre aux systèmes Osek (peut-être une sorte de sous-Pola dédiée aux systèmes Osek), plutôt que de rajouter cette construction de groupe dans le langage ?

4.2.2 partitionnement à la Arinc 653

L'avionique est un domaine possédant les niveaux de criticité les plus élevés : tout logiciel embarqué dans un avion est sujet à une certification qui est régulée par la norme DO-178. Dans le respect de cette norme, la nécessité atemporelle de réutilisation de code, permettant des cycles de développement plus courts, a donné naissance à une architecture logicielle et matérielle adaptée à la problématique avionique : l'IMA, *Integrated Modular Avionics*. Cette architecture définit un réseau de systèmes temps réel répartis dans l'avion. Ces systèmes sont répartis en modules de calcul de criticité différentes. Pour permettre de regrouper des fonctions logicielles différentes dans ces modules, le standard Arinc 653 est utilisé.

Arinc 653 définit une API permettant le partitionnement spatial et temporel des applications. Cela permet de faire cohabiter au sein du même système matériel, des fonctionnalités logicielles complètement étrangères et cela aussi bien en terme d'espace, c'est-à-dire que les ressources sont allouées de manière exclusive, que en terme de temps. Ce découpage temporel et spatial définit des *partitions* dont la configuration est fixée de manière statique, hors ligne. Ainsi on est certain que chaque fonctionnalité logicielle va recevoir périodiquement la même quantité de temps, garantissant la régularité et la réactivité des modules et du système dans sa globalité.

Un système Arinc 653 (voir figure 4.15) est composé d'une couche système basse, chargée de

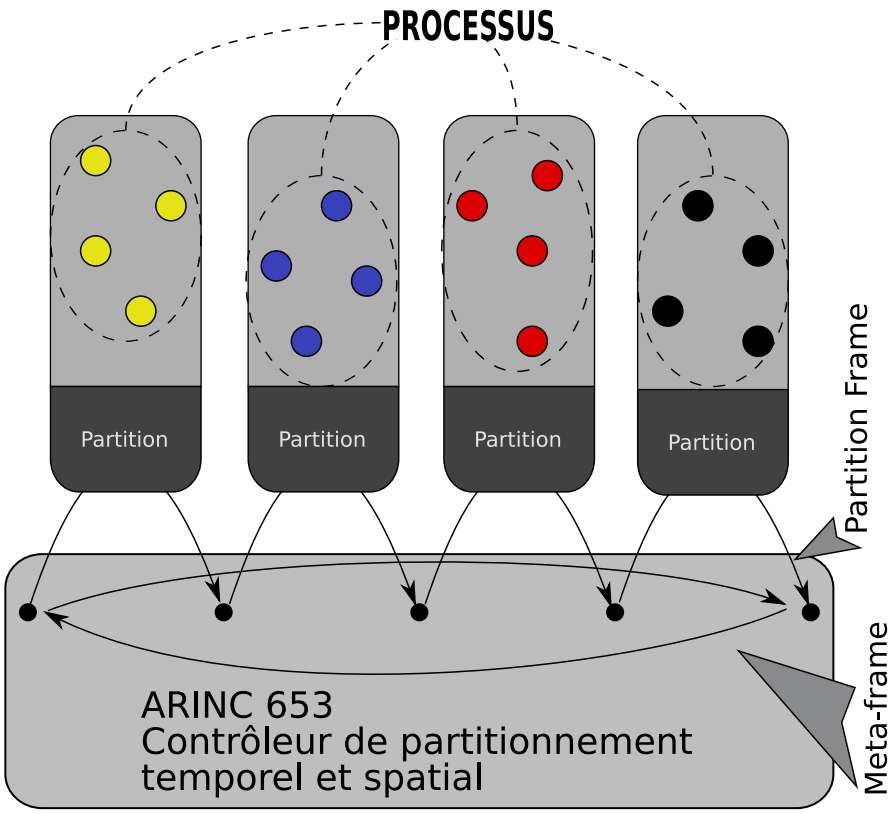


Fig. 4.15 – Partitionnement dans Arinc 653

contrôler (autrement dit, d'ordonnancer) la séquence périodique des partitions. Ce système contrôleur est très rigide : seules quelques valeurs, notamment la durée d'une partition, peuvent être configurées à ce niveau, mais en plus la reconfiguration ne peut s'effectuer qu'une fois l'avion au sol (et après avoir obtenu la nouvelle certification).

La couche haute, que l'on pourrait quasiment appeler "applicative", de Arinc 653 comprend des partitions. Ces partitions délimitent aussi bien un espace mémoire qu'une fenêtre temporelle pendant laquelle la partition va s'exécuter de manière exclusive. Une partition est définie à la fois par un mécanisme de bas niveau permettant essentiellement la spécification de la ou les politiques d'ordonnancement en vigueur sur la partition ; mais également par un ensemble de tâches, appelées *processus*.

L'intervalle de temps alloué à l'exécution d'une partition est appelée fenêtre (ou *frame*). Le séquencement des partitions par le contrôleur de bas-niveau est périodique et la période est appelée méta-fenêtre (ou *meta-frame*). La figure 4.16 donne un exemple d'exécution d'un système Arinc 653. Il est à noter que d'une *meta-frame* à l'autre, le comportement d'une partition peut changer (mais sa durée d'exécution, elle, reste constante). Il est possible que tous les processus d'une partition ne puissent s'exécuter dans le temps alloué à la partition, comme c'est le cas pour la deuxième partition de l'exemple. Dans ce cas, la partition dans son ensemble est suspendue, gardant les processus dans l'état dans lequel ils se trouvaient lors de la suspension. Une fois que le contrôleur de partition a redonné la main à la partition, l'exécution des processus est reprise.

A la phrase précédente, nous aurions voulu ajouter "comme si de rien n'était". Seulement, le standard définit que le temps connu par les partitions est global. C'est-à-dire que dans le cas du processus interrompu par le contrôleur de partition, lors de sa reprise, celui-ci est capable de savoir s'il a été interrompu en prenant connaissance du temps écoulé.

Ceci implique que toute référence au temps est globale et ce, même au sein d'une partition. Cela est certes puissant, puisqu'on a ainsi la connaissance du temps jusqu'au niveau applicatif. Mais ceci a un effet pervers : toutes ces références peuvent tomber en dehors de la partition. Ainsi, est-il acceptable de considérer un paramètre de période d'une tâche qui "tombe" en dehors de la partition ? Puisque

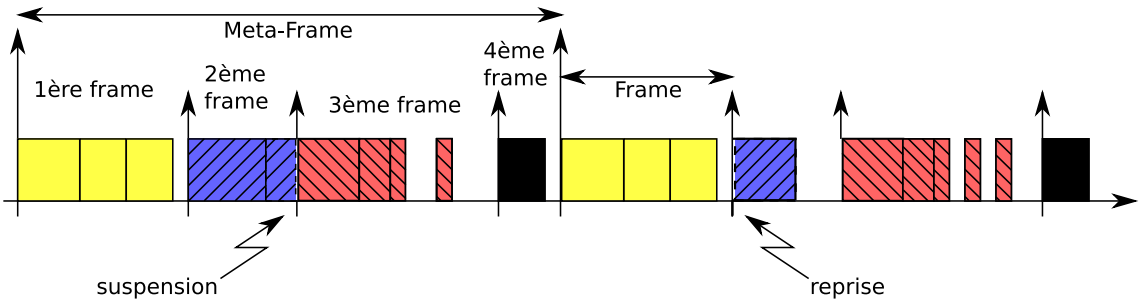


Fig. 4.16 – Exemple d'exécution ARINC 653

la partition n'est alors pas activée, aucune réaction n'est possible et le comportement du système est équivalent avoir à celui dans lequel on aurait allongé la durée de la partition jusqu'à la prochaine activation de la partition. Il y a plus grave : l'échéance peut aussi tomber en dehors de la partition. Le traitement de cette erreur ne pourra pas être effectuée avant la prochaine instance de la partition. Pourquoi l'utilisateur a-t-il spécifié un instant qu'il devrait savoir inaccessible à la partition ? Quel sens donner à une telle spécification ?

L'intérêt d'utiliser le temps global nous semble bien faible en comparaison des potentialités d'erreurs que son utilisation introduit. De plus, à notre connaissance, l'usage industriel est de restreindre l'utilisation de Arinc 653 en imposant notamment de pouvoir faire contenir l'exécution de toutes les tâches critiques sur une seule fenêtre de partition. Ainsi l'échéance de ses tâches est la fin de la fenêtre temporelle de la partition et leur période est égale à la méta-fenêtre. Seules les tâches ne possédant pas de contraintes temps réel peuvent être réparties sur plusieurs fenêtres. Cette restriction implique que la modélisation de ces systèmes est équivalente, à une traduction des périodes près, que l'on considère le temps global ou bien local à la partition.

De cette hypothèse dépend la manière de modéliser le système. Si l'on accepte de modéliser le système sous hypothèse de temps local, alors il est possible d'associer les partitions à des systèmes préemptables en Pola. Dans le cas général, si l'hypothèse locale n'est pas acceptée, il n'est pas possible de faire cette association système/partition et le découpage doit être fait autrement.

Hypothèse de temps local à la partition

Lorsque le temps est local à la partition, il est possible de suspendre toutes les horloges de la partition et ainsi de coder une partition par un système préemptif Pola.

Le modèle donné par la figure 4.17 fait usage du caractère préemptif des systèmes. Il est composé de quatre systèmes : *arinc, partition1, partition2* et *tachesdefond*.

Le premier système, nommé *arinc*, spécifie le comportement du contrôleur de partition. Il active alternativement les deux partitions. La première partition dure 30 unités de temps, tandis que la seconde prend 20 unités de temps. Son comportement est très basique, c'est pourquoi nous l'avons entièrement écrit à l'aide du langage de comportement (utiliser toute la mécanique de Pola eût été excessivement verbeux).

Viennent ensuite les deux partitions Arinc 653, modélisées à l'aide de deux systèmes déclarés préemptifs (**preemptable**). Le système *partition1* est initialement actif, au contraire du système *partition2*, son inactivité initiale étant signalée par **noinit**. Le système *partition1* est constitué d'une tâche de haute priorité préemptant systématiquement une deuxième de plus faible priorité. La deuxième tâche, *t2*, admet pour période le temps d'exécution de sa partition, elle est donc exécutée une seule fois par fenêtre de *partition1*, tandis que la tâche plus prioritaire *t1* est exécutée trois fois pendant la fenêtre temporelle réservée à la partition.

La deuxième partition, *partition2*, comprend deux tâches dont les périodes sont égales à la durée de la partition et dont la capacité cumulée est égale à la période. La deuxième partition est ainsi entièrement utilisée. La deuxième tâche est un peu spéciale car c'est un serveur de tâche permettant à une autre tâche, ne faisant pas partie du système *partition2*, de s'exécuter sans que le système ne

```
system arinc is
  behavior is
    tr t1 [30,30] p1 -> p2
    tr t2 [20,20] p2 -> p1
    lb partition1.active p1
    lb partition2.active p2
end
preemptable system partition1 is
  res proc is preemptable
  task t1 is
    action a1 in [8,8] with alloc1
    period [10,10]
    deadline 8
    policy RM
  end
  task t2 is
    action a1 in [6,6] with alloc1
    period [30,30]
    deadline 30
    policy RM
  end
  allocation alloc1 is
    resources proc
    tasks t1,t2
  policy RM is min P
end
noinit preemptable system partition2 is
  res proc is preemptable
  task t1 is
    action a1 in [14,14] with alloc1
    period [20,20]
    deadline 20
    policy CM
  end
  task t2 is
    action demarreTDF in [1,1] with alloc1
    action stoppeTDF in [5,5] with alloc1 giveback endoftask
    period [20,20]
    deadline 20
    policy CM
    behavior is
      tr t1 p1 -> p2
      tr t2 p2 -> p1
      pl p1 (1)
      lb partition2.t2.demarreTDF t1
      lb partition2.t2.stoppeTDF t2
      lb tachesdefond.active p2
  end
  allocation alloc1 is
    resources proc
    tasks t1,t2
    policy CM is max C
end
noinit preemptable system tachesdefond is
  res proc is preemptable
  task t1 is
    action a1 in [20,20] with alloc1
    period [20,20]
    policy RM
  end
  allocation alloc1 is
    resources proc
    tasks t1
  policy RM is min P
end
```

Fig. 4.17 – Un système utilisant le mécanisme de partitions Arinc 653

sache, ou n'ait à savoir, quoi que soit sur la tâche "servie".

La tâche *t2* est composée de deux actions : une pour lancer le serveur et une autre pour attendre la fin de la tâche cliente ou pour l'interrompre (au sens de la suspension/reprise) si elle n'est pas terminée.

La tâche cliente est donnée par le système *tachedefond* contenant une tâche qui s'exécute sans arrêt. Pour terminer la tâche cliente du système *tachesdefond*, la tâche serveur doit être exécutée quatre fois.

Le comportement que nous venons de commenter est représenté par le chronogramme de la figure 4.18.

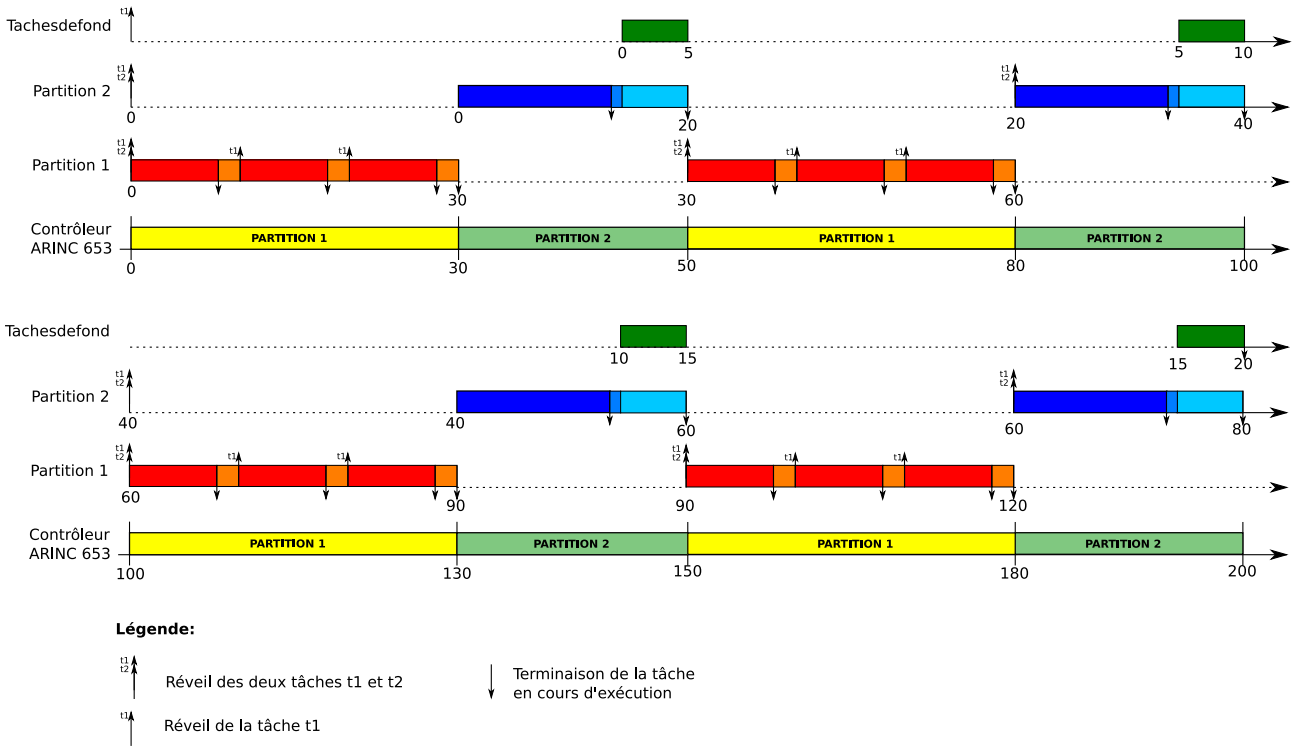


Fig. 4.18 – Chronogramme du comportement du modèle de la figure 4.17

Hypothèse de temps global au système

Comme nous l'avons vu dans la section précédente, l'hypothèse de temps local à une partition s'adapte bien au langage Pola car une partition est alors modélisée par un bloc système préemptif. Si cette hypothèse de localité temporelle n'est pas valide, il n'est plus possible d'utiliser de systèmes préemptifs, ni même de systèmes. En effet, sous une hypothèse de temps global au système, il n'est plus souhaité de suspendre la progression des horloges des partitions alors que celles-ci ne sont plus actives. De plus, dans le cas où l'on n'utilise pas de systèmes préemptifs, lorsque le système est désactivé *toutes* les horloges du système le sont également, ce qui a pour effet de les remettre à zéro une fois le système activé à nouveau.

Puisqu'il n'est plus possible de spécifier les partitions dans des systèmes séparés, nous allons voir comment tout regrouper au sein d'un seul système de telle sorte que ce regroupement soit le plus généralisable possible. Ce qui suit décrit une méthode de transformation depuis un système utilisant un temps local aux partitions en un système utilisant l'hypothèse de temps global, démontrant par là l'applicabilité de notre langage et cela même sur des domaines qui nous semblent potentiellement dangereux en terme de spécification. L'applicabilité de cette transformation est néanmoins sujette à une condition : le séquencement des partitions au cours d'une *meta-frame* doit être tel qu'il n'y a plusieurs occurrences d'une même partition dans le séquencement (autrement dit, les éléments partitions participant à la séquence de la *meta-frame* est un ensemble (et non un multi-ensemble)). Dans le cas contraire, la transformation devient complexe et nous doutons fortement de l'intérêt même d'une telle transformation. Le système nous semble reposer sur des bases suffisamment différentes pour nécessiter d'être repensé par rapport à cette nouvelle perspective plutôt que d'être adapté.

Le contrôleur est modélisé par une tâche la plus prioritaire du système possédant x actions. Chacune de ces actions représente une des x partitions et utilise une allocation différente allouant à la tâche contrôleur la liste de ressources représentant les $x-1$ autres partitions. Par exemple, pour exécuter la première action, c'est-à-dire pour exécuter la première partition, la tâche *contrôleur* doit posséder les ressources représentant les partitions 2 à x. Un processus appartenant à la partition i est modélisé par une tâche ayant besoin de la ressource associée à la $i^{ème}$ partition. Puisque le

contrôleur est la tâche la plus prioritaire et possède toutes les ressources des partitions, sauf celle en cours d'exécution, alors seuls les processus de cette partition peuvent s'exécuter.

Traduction de la périodicité. Pour mener à bien cette traduction nous aurons à recalculer les périodes de telle façon que l'on ne tombe plus en dehors de la partition. Dans la suite, nous utiliserons les termes suivants :

- p, désigne la période de la tâche en hypothèse locale que l'on souhaite transformer de façon à pouvoir opérer au sein d'un système utilisant l'hypothèse globale.
- par, désigne la durée de la partition sur laquelle s'exécute la tâche.
- mf, désigne la *meta-frame*.

Une tâche est divisée en autant d'instances que nécessaire, c'est-à-dire

$$\text{nombres d'instances} = \frac{ppcm(p, par)}{p} \tag{4.4}$$

Il va de soi que cette technique n'est applicable de manière pratique que si le résultat de cette division est petit. Le terme petit est subjectif et dépendra du lecteur. Néanmoins il est assuré qu'il existe une valeur (qui n'est peut-être pas si "petite" que cela) rendant la traduction impraticable.

Le ppcm (plus petit commun multiple) représente le temps (sous l'hypothèse locale) nécessaire pour revenir dans la configuration initiale. Cela correspond donc à la méta-période de l'ensemble des instances de la tâche par rapport aux instants de démarrage de la partition. Cette méta-période est à étendre en prenant compte le temps d'exécution des autres partitions pour transformer ces grandeurs en accord avec l'hypothèse globale. Cela donne :

$$\text{période} = \frac{ppcm(p, par)}{p} \times mf \tag{4.5}$$

Chaque décalage o_i de l'instance i, sans tenir compte des *offsets*, ni des partitions intervenant avant, est calculé par :

$$o_i = \lfloor \frac{i \times p}{par} \rfloor \times mf + ((i \times p)\%par) \tag{4.6}$$

Où $\lfloor x \rfloor$ donne la partie entière arrondie à l'entier inférieur de x et % est l'opération modulo qui retourne le reste de la division entière. Ainsi $\lfloor \frac{i \times p}{par} \rfloor$ donne le résultat de la division entière de $\frac{i \times p}{par}$ et représente le nombre de fois où la partition est activée avant que l'instance i ne soit réveillée. Multiplié par la durée totale de la *meta-frame* et en n'oubliant pas le reste de la division, on obtient la valeur de l'offset de l'instance i.

Si un offset était initialement spécifié pour la tâche que l'on est en train de décomposer, il doit être systématiquement ajouté à l'équation précédente. Mais cet ajout n'est possible qu'après sa transformation :

$$o = \sum_k par_k + \lfloor \frac{O}{par} \rfloor \times mf + (O\%par) + o_i \tag{4.7}$$

Où o est la nouvelle valeur de l'offset et O l'ancienne. L'expression $\sum_k par_k$ représente la somme des durée des k partitions intervenant avant la partition courante. Mise à part cette somme, l'équation est similaire à la précédente et est réutilisable pour le calcul de la nouvelle échéance :

$$d = \sum_k par_k + \lfloor \frac{O + D + (i \times p)}{par} \rfloor \times mf + ((O + D + (i \times p))\%par) \tag{4.8}$$

Où d est la nouvelle valeur de l'échéance et D l'ancienne.

La capacité ne subit pas de transformation lors du passage à l'hypothèse de temps global.

La politique constitue également un élément qu'il faut transformer. Puisque nous mélangeons plusieurs partitions au sein d'un même système, le moyen le plus simple de définir la nouvelle politique générale est d'utiliser les niveaux de priorité arbitraires, définis grâce au mot-clé **level**. Le contrôleur de partition possède la priorité la plus importante. La politique de chacune des partition est également

transformée en niveaux de priorités en prenant garde que le contrôleur reste plus prioritaire que toutes les tâches des partitions. Comme les partitions sont exécutées de manière exclusives, les tâches de partitions différentes ne peuvent se faire concurrence.

Le modèle de la figure 4.19 est une démonstration de cette transformation, appliquée au modèle de la figure 4.17.

Comme nouveauté, on peut noter l'utilisation de **noinit** en face d'allocations : celles-ci ne sont pas initialement actives. L'activité d'une allocation possède deux niveaux : l'activité globale de l'allocation donnée par *arinc.allocation.active*, qui est le seul niveau désactivable par un mot-clé comme **noinit**, et l'activité de l'allocation par rapport à une tâche *t arinc.allocation.t.active*. Même si l'allocation *ctrlP2*, par exemple, est désactivée globalement par **noinit** (puisque c'est la seule façon de la désactiver), comme nous n'aurons pas à modifier l'activité de ces parties, il est indispensable de les activer : c'est ce qui est fait dans le comportement de la tâche *controleur*. L'activité d'une partie de l'allocation dépend avant tout de son activité globale, ainsi même si on réactive la partie liée à la tâche *controleur* (en fait la seule), l'allocation n'est pour autant pas active de manière globale.

Une autre nouveauté est l'utilisation des niveaux de priorités arbitraires (**level**). La politique *polARINC* définissant les tâches les plus prioritaires comme étant celles qui possèdent le niveau le plus bas (**policy** polARINC **is** min L).

Le modèle aurait pu être plus gros si nous avions scrupuleusement suivi les règles de transformation. En effet, dans ce cas, la tâche *TDFt1* aurait du être découpée en quatre, avec une période de 50 unités de temps. Mais comme c'est la seule tâche du système *tachesdefond*, on peut simplifier en utilisant la période de ce qui aurait dû être quatre instances, c'est pour cela que sa période est fixée à 200.

4.3 Conclusion

Nous avons présenté dans ce chapitre un langage dédié à la spécification des systèmes de tâches temps réel. Ce langage a le double objectif de permettre de définir un périmètre d'application des techniques de vérification et de fournir aux experts de la conception des systèmes temps réel un outil permettant de cacher toute la complexité de la technique automatique de vérification par *model checking*.

Après avoir défini les types de systèmes que nous souhaitions prendre en compte, nous nous sommes concentrés sur des cas d'études qui nous semblaient représentatifs des besoins dans le milieu aéronautique et automobile. Nous avons donné des spécification Pola modélisant ces études de cas : d'une manière générale, le langage se comporte bien face à ce genre de problématique, mais si certaines améliorations pourraient être apportées comme nous l'avons vu pour le cas d'étude Osek et que certaines hypothèses de gestion du temps s'adaptent moins bien à notre langage, comme nous l'avons vu dans le cas d'Arinc 653. Néanmoins, nous pensons, que contrairement à d'autres problématiques que notre langage gère mal (comme les protocoles à héritages de priorité), la mauvaise gestion du temps global pour Arinc 653 vient de ce que mécanisme permet l'expression de système dont la complexité est très grande, impliquant du coup que le modèle est également complexe.

Si ces cas d'études permettent d'éprouver notre DSL, d'autres caractéristiques n'ont pas été exposées : il n'a pas été question de systèmes multi-ressources, alors que Pola permet la spécification de manipulations de ressources qui dépassent souvent ce que les outils actuels que nous connaissons peuvent gérer.

D'autres mécanismes, comme les protocoles à héritages de priorités ne sont pas (au pire) ou mal pris (au mieux) en compte. Mais nous pensons que cette difficulté ne vient pas de difficulté intrinsèque de ces protocoles, mais plutôt d'un manque au niveau du mécanisme d'allocation, qu'il conviendrait de généraliser pour permettre de prendre en compte ces protocoles.

```
system arinc is
  res partition1 is preemptable
  res partition2 is preemptable
  res tachedefond is preemptable
  task controleur is
    action a1 in [30,30] with ctrlP1 giveback
    action a2 in [20,20] with ctrlP2 giveback
    level 1
    policy polARINC
    behavior is
      tr t1 p1 -> p2
      tr t2 p2 -> p1
      pl p3 (1)
      lb arinc.controleur.a1 t1
      lb arinc.controleur.a2 t2
      lb arinc.ctrlP1.active p1
      lb arinc.ctrlP2.active p2
      lb arinc.ctrlP2.controleur.active p3
      end
  allocation ctrlP1 is
    resources partition2, tachedefond
    tasks controleur
  noinit allocation ctrlP2 is
    resources partition1
    tasks controleur
  task p1t1_1 is
    action a1 in [8,8] with P1alloc
    period [50,50]
    deadline 8
    level 2
    policy polARINC
  end
  task p1t1_2 is
    action a1 in [8,8] with P1alloc
    period [50,50]
    offset [10,10]
    deadline 18
    level 2
    policy polARINC
  end
  task p1t1_3 is
    action a1 in [8,8] with P1alloc
    period [50,50]
    offset [20,20]
    deadline 28
    level 2
    policy polARINC
  end
  task p1t2 is
    action a1 in [6,6] with P1alloc
    period [50,50]
    deadline 50
    level 3
    policy polARINC
  end
  allocation P1alloc is
    resources partition1
    tasks p1t1_1, p1t1_2, p1t1_3, p1t2
  allocation P2alloc1 is
    ressources partition2, tachedefond
    tasks p2t1, p2t2
  noinit allocation P2alloc2 is
    ressources partition2
    tasks p2t2
  task p2t1 is
    action a1 in [14,14] with P2alloc1
    period [50,50]
    deadline 50
    offset [30,30]
    level 4
    policy polARINC
  end
  task p2t2 is
    action demrTDF in [1,1] with P2alloc1 giveback
    action stopTDF in [5,5] with P2alloc2 endoftask
    period [50,50]
    offset [30,30]
    deadline 50
    level 5
    policy polARINC
    behavior is
      tr t1 p1 -> p2
      tr t2 p2 -> p1
      pl p3 (1)
      lb arinc.p2t2.demrTDF t1
      lb arinc.p2t2.stopTDF t2
      lb arinc.P2alloc1.active p1
      lb arinc.P2alloc2.active p2
      lb arinc.P2alloc2.p2t2.active p3
  end
  allocation TDFalloc is
    resources tachedefond
    tasks TDFt1
  taskTDFt1 is
    action a1 in [20,20] with TDFalloc
    period [200,200]
    level 6
    policy polARINC
  end
  policy polARINC is min L
end
```

Fig. 4.19 – Modèle de la fig. 4.17 transformé pour l'hypothèse globale

Chapitre 5

Sémantique de Pola

Toujours dans notre objectif de donner un sous ensemble connu des systèmes de tâches temps réel vérifiables à l'aide des *ipTPN*, il convient de donner la sémantique du langage Pola par une traduction dans ce formalisme. Pour mener à bien cette traduction, il faut pouvoir manipuler aussi bien les concepts du langage source, Pola, que ceux du langage de destination, les *ipTPN*. Pour ce faire, nous allons avoir besoin d'extraire les données du langage Pola et d'exprimer ces données à l'aide des réseaux de Petri. Nous introduirons donc dans un premier temps un méta-modèle de notre langage qui nous servira de base afin d'exprimer précisément quels éléments du langage sont traduits et/ou quelles informations du modèle sont nécessaires à la traduction d'un élément. Dans un deuxième temps, nous introduirons un langage qui nous permettra d'exprimer une génération conditionnelle (ou paramétrique) des réseaux de Petri.

5.1 Le méta-modèle Pola

La *syntaxe abstraite* d'un langage peut être donnée sous la forme d'un méta-modèle. Un méta-modèle donne les règles de construction d'un modèle. Un modèle issu du méta-modèle donne donc la syntaxe abstraite d'un «programme» du langage. Un tel «programme» est également un modèle (mais du langage plutôt que du méta-modèle) : il se distingue du modèle du méta-modèle par sa *syntaxe concrète*. Une syntaxe abstraite est utile pour exprimer les concepts fondamentaux du langage et leurs liens, sans aller au niveau de précision inutile de la syntaxe concrète.

Le méta-modèle du langage Pola de la figure 5.1 est donné par un langage graphique (Ecore) qui peut être vu comme un sous-ensemble du modèle de classes Uml et dont nous allons maintenant donner une brève introduction.

Les rectangles sont des *classes* d'objets, ce sont les éléments de base manipulables du langage. Le nom des classes est donné par le premier champ en **gras**. En dessous de ce nom sont définis les *attributs*. Tout de suite après le "+" vient le nom de l'attribut puis, après les ":" vient son type. Les arcs terminés par des losanges signifient que la classe qui est à l'extrémité sans losange est une partie de la classe qui est à l'autre bout (celui avec le losange). Par conséquent, un système est constitué de ressources, de tâches, d'allocations, etc. L'indication accolée à l'arc précise les bornes sur le nombre de classes possibles, où $a..b$ représente l'intervalle $[a, b]$ et $a..*$, l'intervalle $[a, \infty[$. Ainsi, le 0..1 indiqué sur l'arc entre la classe *Deadline* et la classe *Task* indique qu'une tâche ne peut avoir qu'un seul élément *deadline* ou pas de *deadline* du tout.

Les types définis dans ce méta-modèle sont les suivants :
- String, l'ensemble des chaînes de caractères ;
- Nat, l'ensemble des entiers naturels ($\mathbb{N}$) ;
- Bool, l'ensemble des booléens ($\mathbb{B}$) ;
- Petri net, l'ensemble des réseaux de Petri ;
- Label List, l'ensemble des listes d'étiquetage, qui sont des couples liant une chaîne de caractère (l'étiquette) à un élément place ou transition d'un réseau de Petri ;

– INTERVAL, l'ensemble des intervalles, potentiellement non bornés, dont les bornes sont dans l'ensemble des entiers naturels et peuvent être strictes. Les intervalles suivants sont tous valides : $[0,1]$, $]0,1[$, $]2,\infty[$;
– Allocation, l'ensemble des classes Allocations ;
– expr, l'ensemble des expressions définissant une politique d'ordonnancement

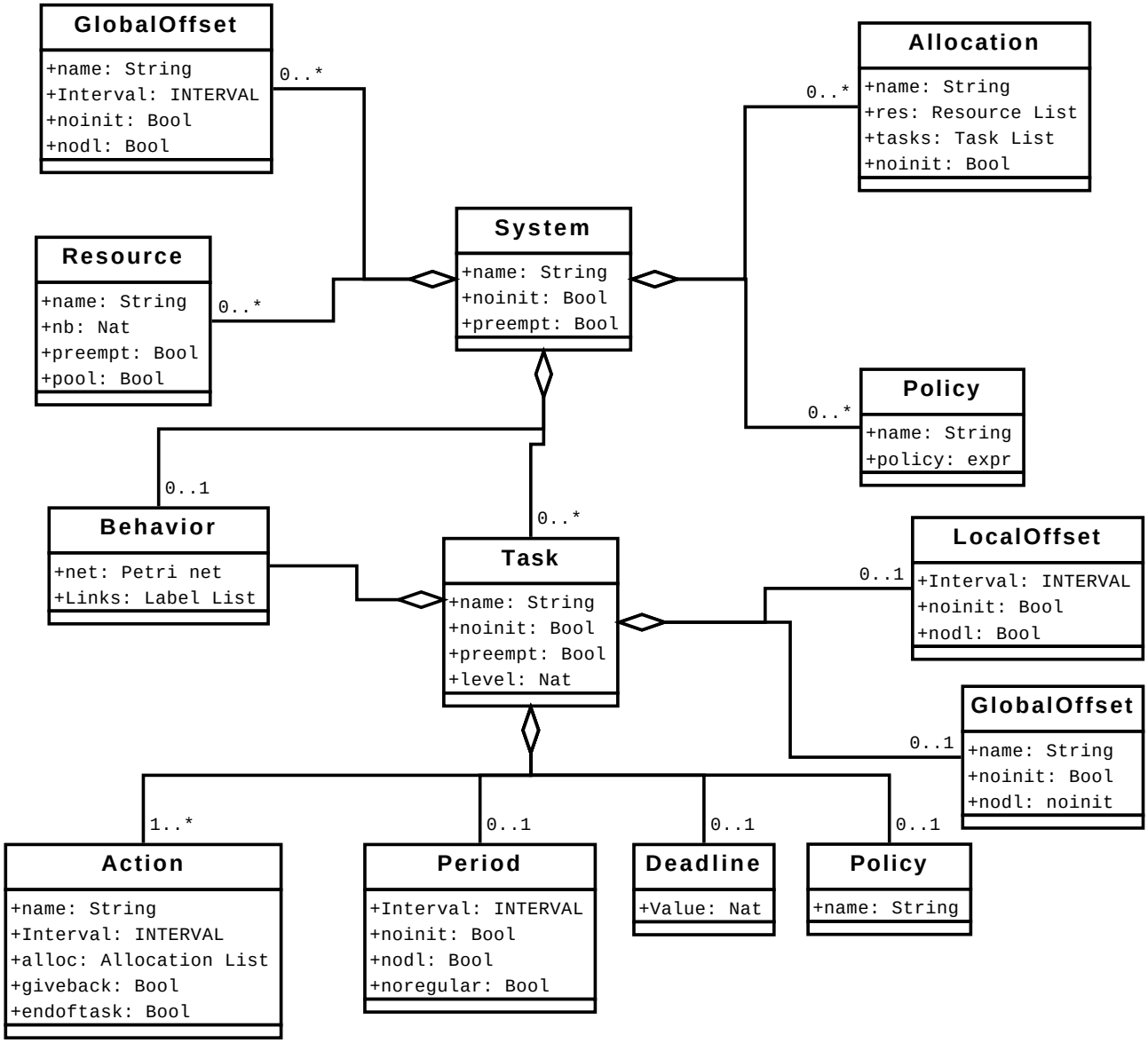


Fig. 5.1 – Le méta-modèle du langage Pola

Nous allons par la suite utiliser des notations se basant sur ce méta-modèle, avec quelque raccourcis que nous présentons maintenant. Pour accéder aux différents éléments du méta-modèle, nous utiliserons la notation pointée usuelle lorsque l'on manipule des objets, ainsi *System.Task.Deadline* permet d'accéder à la classe *Deadline* accessible par *Task* depuis la classe *System*. On confondra l'attribut *name* d'une classe et la classe elle-même. Lorsqu'il n'est pas obligatoire qu'une classe possède une instance dans le modèle, comme la classe *Deadline* par exemple, nous utiliserons la valeur $\perp$ afin de tester l'existence de l'instance. En ce qui concerne la classe *Deadline*, on peut donc tester l'existence d'une de ces instances par $t.Deadline{=}\perp$, où t est une instance de la classe *Task*.

Afin de réduire la verbosité, pénalisant souvent la lisibilité, le nom d'une classe c pourra être écrit en majuscule, pour signifier le nom absolu (en notation pointée) de l'instance de la classe c, parent de

la classe de l'objet en cours de traduction. Par exemple, si l'on traduit une action a, et si l'attribut *name* de l'instance de la classe père *Task* de l'action a vaut t, et si l'attribut *name* de l'instance de la classe grand-père de l'action a vaut s, alors on a TASK = "s.t".

5.2 Préprocesseur de réseaux de Petri

Nous avons maintenant les outils adéquats pour une exploration du modèle Pola. Il nous reste à nous doter des outils pour la génération du réseau de Petri correspondant. La façon de faire la plus directe serait de donner un algorithme, qui selon les cas générerait tel ou tel élément du réseau de Petri. Nous ne pensons pas que cette façon de faire convienne à une présentation compréhensible et agréable. Nous allons plutôt ajouter une couche "générative" aux réseaux de Petri. Dans un premier temps de manière textuelle, ce qui a le grand avantage d'être concis, mais possède l'inconvénient d'être difficile à lire. Nous procéderons donc dans un deuxième temps à une adaptation de cette écriture textuelle à une écriture graphique, qui nous l'espérons sera plus agréable à lire, car moins condensée, tout en gardant les mêmes informations que la version textuelle.

Il ne faut pas voir cette couche générative comme faisant partie des réseaux de Petri. C'est une écriture permettant de générer des chaînes de caractères ayant du sens : le résultat de cette génération étant la notation textuelle des réseaux de Petri que nous utilisons depuis le début ; ou de générer des éléments graphique des réseaux de Petri de manière paramétrique. C'est donc un préprocesseur que nous comptons définir.

Ce préprocesseur est exprimé à l'aide des séquents de la sémantique opérationnelle. Un séquent $\frac{A}{B}$ signifie que l'on peut inférer B à partir des hypothèses A. Pour décrire la sémantique, nous utilisons la syntaxe $env \vdash expr \Rightarrow r$ qui signifie que dans l'environnement env, l'évaluation de $expr$ produit le résultat r. Dans notre cas, l'environnement est donné par le modèle Pola M et l'élément en cours de traduction e. Nous ne détaillerons pas la sémantique des expressions car elle correspond à celle des expressions de la logique du premier ordre.

En premier lieu, il est possible avec ce préprocesseur d'avoir une écriture conditionnelle du résultat sans alternatives. La syntaxe est (***expression*** $\rightarrow$ ***résultat***). L'expression est une interrogation du modèle et le résultat est ce qui devra être écrit sur la sortie après une passe du préprocesseur. Le résultat n'est écrit que si l'évaluation de l'expression retourne **true**.

$$\frac{(M,e) \vdash expr \Rightarrow \textbf{true} \quad (M,e) \vdash result \Rightarrow r}{(M,e) \vdash (expr \rightarrow result) \Rightarrow r} \tag{5.1}$$

Exemple

Supposons que dans un modèle x, une ressource R soit définie comme préemptable et que l'on soit en train de traduire l'élément t, on a donc $(x,t) \vdash SYS.R.preempt$ qui s'évalue à **true** et en posant que $(x,t) \vdash a$ s'évalue à r, alors l'expression $(SYS.R.preempt \rightarrow a)$ s'évalue à r.

Les deux séquents suivants représentent la sémantique du motif (***expression*** **?e1 :e2)**, reprenant la notation conditionnelle avec alternative du langage C. Les deux séquents qui suivent expriment que, lorsque l'expression est vraie, alors la première des deux alternatives est retournée (premier séquent) ou la seconde, dans le cas contraire (deuxième séquent).

$$\frac{(M,e) \vdash expr \Rightarrow \textbf{true} \quad (M,e) \vdash e_1 \Rightarrow r}{(M,e) \vdash (expr?e_1 : e_2) \Rightarrow r} \tag{5.2}$$

$$\frac{(M,e) \vdash expr \Rightarrow \textbf{false} \quad (M,e) \vdash e_2 \Rightarrow r}{(M,e) \vdash (expr?e_1 : e_2) \Rightarrow r} \tag{5.3}$$

Exemple

En modifiant légèrement l'exemple précédent, **(*not SYS.R.preempt* ?e1 :e2)** s'évaluera à r, où r est le résultat de l'évaluation de e_2, c'est-à-dire $(x,t) \vdash e_2 \Rightarrow r$, puisque $(x,t) \vdash notSYS.R.preempt \Rightarrow$ **false**, c'est-à-dire que dans le modèle x il n'est pas vrai que la ressource R n'est pas préemptable.

Nous aurons aussi besoin d'un itérateur pour parcourir les listes définies dans le méta-modèle. La syntaxe est $(\forall r \in L)(f(r))$, où $f(r)$ est une expression dans laquelle les variables r sont remplacées tour à tour par les éléments de la liste L. Le deuxième séquent gère la fin de l'itération sur la liste L : lorsque sa fin est atteinte (caractérisée par une liste vide), le caractère ϵ, neutre pour la concaténation est retourné.

$$\frac{(M,e) \vdash L \Rightarrow a :: l \quad (M,e) \vdash f(a) \Rightarrow v \quad (M,e) \vdash (\forall r \in l)(f(r)) \Rightarrow s}{(M,e) \vdash (\forall r \in L)(f(r)) \Rightarrow v.s} \tag{5.4}$$

$$\frac{(M,e) \vdash L \Rightarrow [\,]}{(M,e) \vdash (\forall r \in L)(f(r)) \Rightarrow \epsilon} \tag{5.5}$$

Exemple

L'expression $(\forall x \in a.alloc.res)(x)$, où a est une action, donne la liste des noms de ressources utilisées par l'allocation de l'action a. La sortie est assurée par la règle 5.8 définie plus bas.

Ces trois commandes fonctionnent pour toute entrée dont on peut extraire des informations sous forme d'une liste de commandes. Les règles ci-dessous définissent le moteur de lecture de l'entrée.

$$\frac{(M,e) \vdash program \Rightarrow a :: l \quad (M,e) \vdash a \Rightarrow r \quad (M,e) \vdash l \Rightarrow s}{(M,e) \vdash program \Rightarrow r.s} \tag{5.6}$$

$$\frac{(M,e) \vdash program \Rightarrow [\,]}{(M,e) \vdash program \Rightarrow \epsilon} \tag{5.7}$$

Finalement la règle ne traitant pas ce qui est en entrée, qui est utilisée dans tous les autres cas que ceux définis dans les règles ci-dessus. Cette règle est en compétition avec les autres règles, elle doit par conséquent être moins prioritaire qu'elles.

$$(M,e) \vdash text \Rightarrow text \tag{5.8}$$

5.3 Notation graphique

La notation graphique est un peu plus ciblée que sa version textuelle : au lieu de pouvoir appliquer la génération conditionnelle avec alternatives à tous les éléments possibles, nous ne l'appliquerons qu'aux arcs puisque c'est le seul cas nécessaire. La génération itérative et l'expression conditionnelle sans alternatives sont quant à elles plus générales.

5.3.1 Associée aux arcs

Le type d'un arc est modifiable selon une condition avec alternatives, puisqu'avant l'évaluation nous ne connaissons pas son type, nous allons avoir besoin d'un arc "générique", auquel est attaché l'expression conditionnelle avec alternatives. Les types d'arc possibles sont : normal, *read arc*, inhibiteur, chronomètre, chronomètre inhibiteur, dont la représentation graphique est donnée par la figure 5.2.

La notation graphique des arcs génériques, supports des expressions conditionnelles avec alternatives de notre "préprocesseur graphique" est décrite par la figure 5.3.

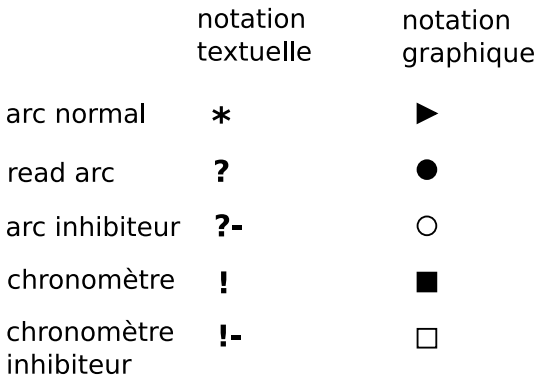


Fig. 5.2 – Les différents types d'arcs, notation graphique et textuelle correspondante

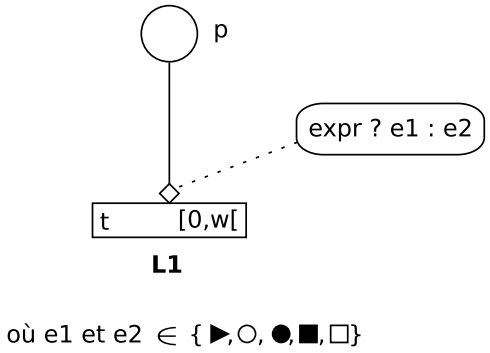


Fig. 5.3 – Un arc générique pour paramétrer le type de l'arc entre p et t

Un arc générique n'a pas de type. C'est une sorte de coquille vide n'ayant aucune existence propre. Mais associé à une expression conditionnelle avec alternatives, l'arc générique voit son type "vide" changer en le type spécifié par l'expression.

5.3.2 Associée à tout élément

L'opérateur d'itération permet la création d'un ensemble de places. Il est exprimé graphiquement par le motif de la figure 5.4. Dans la partie haute du bloc est donné le nom de la liste sur laquelle on veut itérer, chaque élément étant retourné dans la variable indiqué après la flèche. Cette variable est alors utilisable dans la deuxième partie du bloc.

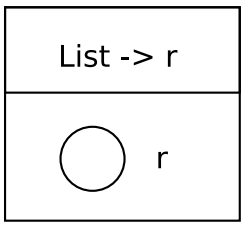


Fig. 5.4 – Un itérateur $(\forall r \in List)(r)$

L'expression conditionnelle sans alternative est graphiquement très proche de sa variante avec alternatives : elle est notée par une expression booléenne contenue dans un rectangle aux angles arrondis et liée à l'élément sur lequel s'applique la condition par un trait en pointillé (voir la figure 5.5).

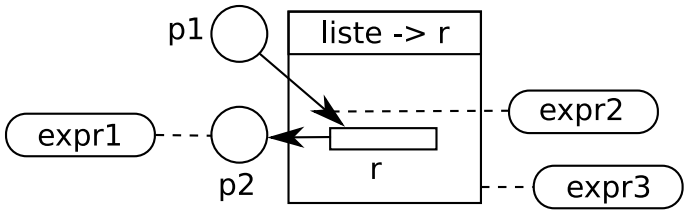


Fig. 5.5 – Des générations conditionnelle sans alternative représentant $(expr3 \rightarrow (\forall r \in liste)(\textbf{tr}\ \text{r}\ (expr2 \rightarrow p1)\ \text{->}\ (expr1 \rightarrow p2)))$

5.4 Sémantique par traduction

5.4.1 Syntaxe textuelle des réseaux de Petri à permission/inhibition

Voici la grammaire du format textuel que nous utiliserons dans la suite :

R	: :=	P | T | INH | PER | ϵ[1]
P	: :=	**“pl”** ID (**“ :”** ID) ? **“(”** NAT **“)”**
T	: :=	**“tr”** ID (**“ :”** ID) ? INTERVAL ? PRECOND* **“->”** NORMAL*
INH	: :=	**“inh”** ID+ (“<” | “>”) ID+
PER	: :=	**“per”** ID+ (“<” | “>”) ID+
ID	: :=	[0-9a-z]+
INTERVAL	: :=	(**“[”** | **“]”**) NAT ((**“,”** NAT (**“]”** | **“[”**)) | **“,w[”**)
NAT	: :=	[0-9]+
PRECOND	: :=	NORMAL | READ | INHIBITOR | SW | SWINH
NORMAL	: :=	(ID **“*”** NAT) | ID
READ	: :=	ID **“ ?”** NAT
INHIBITOR	: :=	ID **“ ?-”** NAT
SW	: :=	ID **“ !”** NAT
SWINH	: :=	ID **“ !-”** NAT

La grammaire débute avec le non terminal R. Chaque déclaration d'un élément du réseau tient sur une ligne : le retour chariot sert de délimiteur de fin.

La déclaration d'une place débute par le mot-clé **pl**, suivi du nom de celle-ci, de l'étiquette éventuelle et finalement du marquage initial. La déclaration d'une place par **pl** est facultative, à moins que celle-ci ne possède un marquage initial différent de 0.

La déclaration d'une transition commence par le mot-clé **tr**, puis vient son nom, son étiquette facultative, son intervalle facultatif, ses préconditions utilisant différents types d'arcs, un signe flèche **->** pour signifier la fin des préconditions et finalement les postconditions. Si aucun intervalle n'est donné, l'intervalle $[0, \infty[$ est utilisé par défaut. A noter qu'un intervalle non borné à droite est noté **w[** : $[0, \infty[$ est donc noté $[0, w[$.

Les deux relations de permissions et d'inhibitions sont déclarées en utilisant respectivement les mots-clés **per** et **inh**. Le signe > ou < indique le sens de la relation. Ainsi, pour déclarer que $t \multimap\!\bullet\ t'$, on pourra aussi bien écrire «**per** t > t'» que «**per** t' < t» et pour déclarer $k \multimap\!\circ\ i$, $k \multimap\!\circ\ j$, $l \multimap\!\circ\ i$, $l \multimap\!\circ\ j$, on pourra écrire «**inh** k, l > i, j».

5.4.2 *system* et ressources

Un élément *system* est pauvre en sémantique puisque c'est un conteneur. Bien que ses paramètres soient constamment consultés par ses différents éléments fils, seule une place est produite, représentant l'activation de l'élément *system*.

Si s est une instance de la classe *System*, alors sa traduction est :

[1] mot vide

pl s.active : s.active **(**(s.noinit ?0 :1)**)**

L'attribut **preemptable** de la classe *System* est traduit, comme nous le verrons plus tard, dans les autres éléments par le type de l'arc testant la valeur de la place *s.active* : si le système est préemptable alors l'arc est un arc à chronomètre et un *read arc* sinon.

Afin de ne pas nous répéter inutilement dans la sémantique des éléments ultérieurs, nous préciserons ici que *toute* transition issue de la traduction des éléments Pola (ce qui exclut donc les transitions de comportement qui ne sont pas synchronisées avec une transition issue de la traduction des éléments Pola) est synchronisée avec la transition suivante :

tr t (s.preempt ? s.active !1 : s.active ?1) ->

Cela signifie que tant qu'un système préemptif est actif toutes les transitions fonctionnent normalement et sont suspendues, prêtes à repartir de l'état dans lequel elles s'étaient arrêtées, lorsque le système devient inactif. Dans le cas où le système n'est pas marqué **preemptable**, le désactiver interrompt et remet à zéro les horloges du système. Un *read arc* est obligatoire pour ne pas remettre à zéro l'ensemble du système à chaque fois que l'un de ses éléments produit un évènement (tire une transition).

De la même façon que la classe *System*, une instance de la classe *Resource* est traduite par une place. Cette place représente le nombre de cette ressource libre dans le système : c'est-à-dire 1 dans le cas par défaut et le nombre de ressources dans le pool (r.pool ici) au cas où la ressource est un *pool* de ressources.

Soit r l'instance de la ressource à traduire.

pl SYS.r : SYS.r **(** (r.pool ? r.nb : 1) **)**

5.4.3 Traduction d'une tâche

L'élément tâche est, comme l'élément *system*, un conteneur qui, au contraire de *system*, ne possède aucune sémantique propre. L'activité d'une tâche est donnée par celle de ses éléments comme le générateur de périodes ou le décalage initial du réveil : il n'y a donc pas de places permettant de repérer l'activité d'une tâche. L'attribut et mot-clé **noinit** qui peut paramétrer une tâche déclare l'inactivité initiale de tous ses éléments.

5.4.4 Traduction d'une action

Soit x une instance de la classe *Action*, soit $a = TASK.x$, la sémantique de l'action x est donnée par :

Algorithme 5 : Sémantique d'une action x

Données : $a = TASK.x$
tr a.tau : a.tau a.Interval TASK.released TASK.dlcheck $(\forall r \in a.alloc.res)(r.preempt?r!1 : r)$ ->
tr a : a [0,w[(a.endoftask ? TASK.released (TASK.deadline $\neq\perp\rightarrow$ TASK.dlcheck) :
TASK.released ?1) ($\forall$r $\in$ a.alloc.res)(a.giveback $\vee$ a.endoftask ? TASK.r : TASK.r ?1) ->
($\forall$r $\in$ a.alloc.res)(a.giveback $\vee$ a.endoftask $\rightarrow$ SYS.r)
inh a.tau > a.tau
per a.tau > a

La figure 5.6 en donne la représentation graphique. Seule la transition *a.tau* porte une réelle information temporelle pertinente. Elle n'est pas tirable, puisqu'elle est définie comme prioritaire sur elle-même. Elle est sensibilisée aux même instants que sa jumelle a et contraint son tir par la relation de permission. Par conséquent, a n'est tirable que dans l'intervalle *a.Interval*. Lorsque l'action termine la tâche, ce qui est signalé par le mot-clé **endoftask**, elle se charge de libérer les ressources auparavant occupées et enlève un jeton de *released*, mettant la tâche en sommeil (si par ailleurs aucun autre évènement n'est intervenu pour réveiller plusieurs fois la tâche). Lorsque une ressource est **preemptable** un arc chronomètre est utilisé permettant la suspension de la tâche.

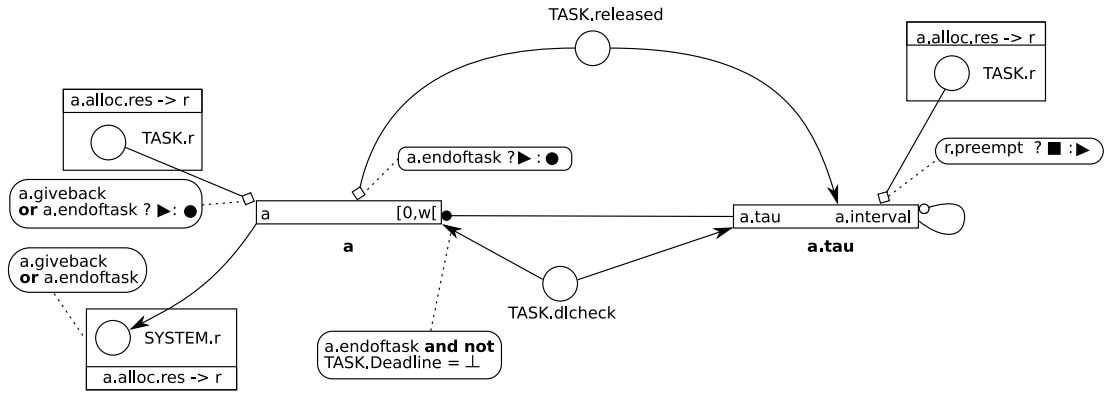


Fig. 5.6 – Motif de traduction d'une action a

5.4.5 Traduction d'une période

Soit x une instance de la classe *Period*, soit p = *TASK.period*, la sémantique du générateur de période est donnée par :

Algorithme 6 : Sémantique d'une période p

Données : p = *TASK.period*
pl p.active : p.active ((noinit ? 0 : 1))
tr p.tempo : p.tau p.Interval p.active ?1 ->
tr p : p [0,w[p.active -> TASK.released (p.noregular → p.active)
(TASK.Deadline ≠⊥→ TASK.dlcheck)
(not (isStrict p.Interval.Max ∨ p.Interval.Max = ∞) →
tr p.max : p.tau [p.Interval.Max,p.Interval.Max] p.active ?1 ->)
inh p.tempo > p.tempo
inh p.max > p.max
per p.tempo > p

La figure 5.7 donne la représentation graphique de la sémantique du motif *period*. La transition $p.max$ admet un intervalle $[p.Interval.Max, p.Interval.Max]$, où l'attribut Max est la valeur de la borne supérieure de $p.Interval$. Cette transition est utile afin de repérer le dernier instant possible pour le tir de p, lorsqu'il existe. Nous verrons dans la section 5.7.2 qu'il est nécessaire que la transition $p.max$ contraigne les allocations par $-\circ$, c'est-à-dire que $p.max$ est une transition *noty* du motif présenté dans la section 5.7.2. Lorsque $p.Interval$ n'est pas borné ($p.Interval.Max = \infty$) ou lorsque sa borne supérieure est stricte (isStrict p.Interval.Max), la transition $p.max$ ne doit pas être générée.

Lorsque la période est **noregular**, cela signifie que l'utilisateur souhaite contrôler lui-même l'activité du générateur. Si la tâche ne possède pas de contrainte temps réel, c'est-à-dire si aucune échéance n'est spécifiée, il faut faire attention à ne pas générer le jeton de demande d'observation du dépassement d'échéance : *TASK.dlchek*.

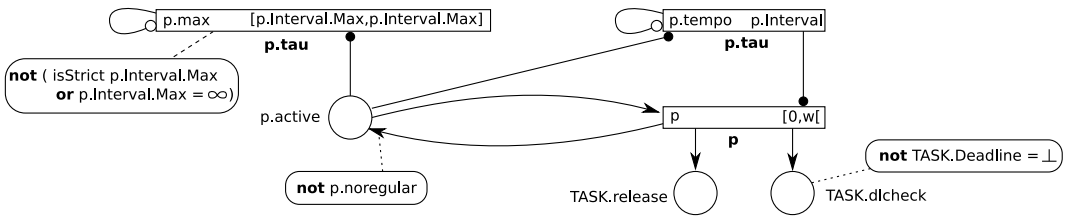


Fig. 5.7 – Motif de traduction d'un générateur de période p

5.4.6 Traduction d'un *offset*

Soit x une instance de la classe *LocalOffset*, soit $o = TASK.offset$, la sémantique du décalage du premier réveil est donnée par :

Algorithme 7 : Traduction d'un *offset* o

Données : $o = TASK.offset$
pl o.active : o.active **(** (noinit ? 0 : 1) **)**
tr o.tempo : o.tau o.Interval o.active ?1 ->
tr o : o [0,w[o.active -> TASK.released TASK.period.active
(TASK.Deadline $\neq\perp\rightarrow$ TASK.dlcheck)
(not (isStrict o.Interval.Max $\vee$ o.Interval.Max $= \infty$) $\rightarrow$
tr o.max : o.tau [o.Interval.Max,o.Interval.Max] o.active ?1 ->)
inh o.tempo > o.tempo
inh o.max > o.max
per o.tempo > o

La figure 5.8 donne la représentation graphique de la sémantique du motif *offset*. Ce motif est très similaire à celui du générateur de période. Les différences sont que, puisque qu'un décalage n'intervient qu'une fois, le tir de o ne remet pas un jeton dans la place *o.active*, mais en rajoute plutôt un dans la place *TASK.period.active*, ce qui a pour effet d'activer le générateur périodique de réveils de la tâche.

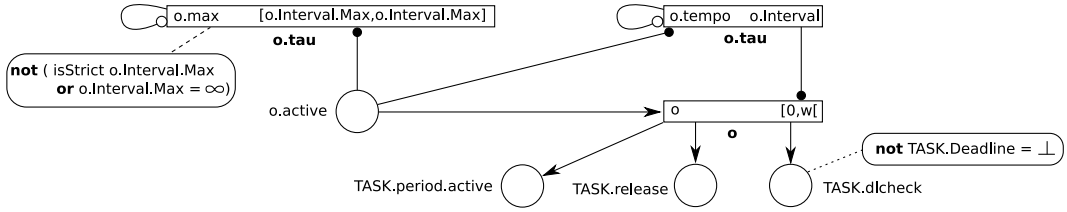


Fig. 5.8 – Motif de traduction d'un *offset o*

5.4.7 Traduction d'une échéance

Soit x une instance de la classe *Deadline*, soit $d = TASK.deadline$, la sémantique d'une échéance est donnée par :

Algorithme 8 : Traduction d'une échéance d

Données : $d = TASK.deadline$
pl TASK.dlcheck : TASK.dlcheck **(** (TASK.noinit ? 0 : 1) **)**
tr d.tempo : d.tempo [d.Value, d.Value] TASK.released ?1 TASK.dlcheck ->
tr d : d [0,w[TASK.released ?1 TASK.dlcheck ->
inh d.tempo > d.tempo
per d.tempo > d

La figure 5.9 donne la représentation graphique de la sémantique du motif de l'observateur d'échéance. Celui-ci n'est initialement actif que lorsque la tâche l'est également. Par la suite, le compteur temporel lié à une échéance tourne tant que la tâche est réveillée (c'est-à-dire tant qu'elle a quelque chose à faire) et si on a demandé de surveiller un potentiel dépassement d'échéance.

5.5 Traduction d'une politique d'ordonnancement

Nous nous limiterons ici à présenter la traduction des politiques d'ordonnancement statiques, la partie dynamique faisant l'objet d'une discussion par la suite.

Donner la sémantique par traduction d'une politique n'est pas aussi direct que dans le cas de l'élément tâche. En fait, l'ordonnanceur est divisé en deux parties : une partie issue de la politique,

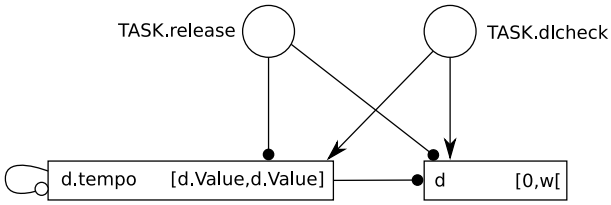


Fig. 5.9 – Motif de traduction d'une échéance d

déterminant quelle tâche est à exécuter avant telle autre ; et l'autre issue des informations de l'élément allocation. Ce que nous allons déterminer ici n'est donc qu'une information partielle, qui n'est pas directement transcriptible en réseaux de Petri, mais qui sera une information utilisée pour ajouter les priorités appropriées aux transitions générées par la traduction du motif allocation.

Ce que nous allons obtenir ici est une relation d'ordre entre les tâches, permettant de connaître les priorités entre chacune d'elles. Comme un système peut comprendre plusieurs politiques, il y aura autant de relations d'ordre que de politiques.

Une politique est donnée par une composition de sous-politiques. Ces sous-politiques sont définies par un opérateur d'ordre sur les entiers naturels, **min** ou **max**, permettant de définir dans quel sens la priorité s'appliquera. La deuxième composante d'une sous-politique est une combinaison linéaire des caractéristiques *capacité* (noté C dans la combinaison linéaire), *période* (P), *échéance* (D) et *niveau arbitraire* (L) d'une tâche. Ainsi, à l'aide de cette combinaison linéaire, on peut associer un entier, donnant le *nombre-priorité* de la politique. Ensuite, on classe ces nombres selon le sens de la priorité : si c'est **min**, alors plus le nombre-priorité est petit, plus la tâche est prioritaire, et inversement, si c'est **max**, alors plus le nombre-priorité est grand, plus la tâche est prioritaire.

La relation d'ordre o que nous utilisons relie des couples de transitions et est définie par : si $(x, y) \in o$, alors x est moins prioritaire que y, abrégé en $x \prec y$. Une politique est donnée par une liste de couples $\langle O, C \rangle$, où $O \in \{min, max\}$ et $C = \alpha C + \beta P + \gamma D + \delta L$ est une combinaison linéaire. Celle-ci peut être évaluée en l'appliquant à une tâche. Soit t une tâche, A l'ensemble de ses actions, p sa période et d son échéance, la valuation de la combinaison linéaire est donnée par $C(t) = \alpha \sum_{a \in A} a.Interval.Min + \beta\ p.Interval.Min + \gamma\ d.Value + \delta\ t.level.$

Les caractéristiques C et P sont des intervalles. Or se baser sur des valeurs indéterminées n'est pas cohérent avec le fait de spécifier une politique statique. Afin de prévenir une telle incohérence, il est possible de soit faire le choix d'interdire l'utilisation de caractéristiques indéterminées dans une politique, soit de prendre arbitrairement une valeur particulière dans cet intervalle de valeurs, auquel cas la borne minimale semble la plus adaptée des valeurs car forcément valide (alors qu'il peut ne pas y avoir de borne finie maximale). Nous avons fait le choix dans l'implémentation de notre outil de ne pas accepter la seconde possibilité : nous interdisons donc purement et simplement l'utilisation de caractéristiques indéterminées dans la politique.

L'algorithme 9 donne la manière dont une sous-politique sépare les tâches qui ne sont pas du même niveau de priorité et range les classes dans la relation d'ordre o. En entrée est donnée une partition de l'ensemble des tâches utilisant la politique pour laquelle on veut donner une sémantique. Pour chacunes des parties de la partition, on calcule le nombre de niveaux que l'on va avoir après application de la sous-politique sur cette partie (ligne 1). Ce nombre représente le nombre de sous-ensembles dans lequel va être découpée la partie s_p que l'on analyse. Ces ensembles sont ensuite crées et stockés dans s_{s_p} (ligne 2). On met alors à jour la relation d'ordre (ligne 3). On regarde pour cela chaque combinaison de couples de chaque parties s_p. Si les deux tâches du couple admettent un nombre-priorité différent alors, selon l'opérateur **min** ou **max** choisi pour la sous-politique, soit $t \prec u$, soit $u \prec t$ est ajouté à o. Le résultat partiel est alors augmenté des nouvelles parties $s^j_{s_p}$, obtenus à partir de s_p (ligne 4).

Algorithme 9 : application d'une sous politique à une partition

Données : Un opérateur $op \in \{min, max\}$;
une combinaison linéaire C;
une partition;
une relation d'ordre partiel o
Résultat : Une partition et o par effet de bord
pour tous les *ensembles* s_p *de la partition* **faire**
1 $N_{s_p} = \{x \in \mathbb{Z} | (\forall t \in s_p)(C(t) = x)\}$
2 $(\forall j \in N_{s_p})(s^j_{s_p} = \{t \in s_p | C(t) = j)\}$
3 $(\forall t \in s_p)(\forall u \in s_p \setminus \{t\})$
$(C(t) < C(u) \Rightarrow$ **si** (op = **min**) **alors** $u \prec t$ **sinon** $t \prec u$
$\wedge C(t) > C(u) \Rightarrow$ **si** (op = **min**) **alors** $t \prec u$ **sinon** $u \prec t)$
4 resutat ← result $\bigcup_{j \in N_{s_p}} \{s^j_{s_p}\}$
fin
retourner *resultat*

Une politique est, de manière générale, constituée de plusieurs sous-politiques, composées par le mot-clé **orelse**. Afin d'obtenir la relation d'ordre pour la politique, il convient d'appliquer l'algorithme 9 pour chacune des sous-politiques. De plus, il est possible de n'avoir pas qu'une seule politique et il faut donc encore appliquer le processus précédemment énoncé à toutes ces politiques, c'est ce qui est accompli par l'algorithme 10. Pour chaque politique du système, l'ensemble S des tâches utilisant la politique est crée (ligne 1), puis la partition est initialisée à l'ensemble comprenant le seul ensemble (la seule partie) S (ligne 2). Ensuite il suffit d'appliquer chacunes des sous-politiques à l'état courant de la partition (ligne 4).

Algorithme 10 : création de la relation d'ordre de la politique

Données : La liste des politiques du système : L_p;
La liste des tâches du système : $Tasks$
Résultat : La relation d'ordre o, par effet de bord
pour chaque *politiques* p *de* L_p **faire**
1 $S = \{t \in Tasks | t.policy = p\}$;
2 $partition = \{S\}$;
3 **pour chaque** *sous-politiques* (Op, C) *de* p **faire**
4 partition ← appliqueSousPolitique Op C partition
fin
fin

En guise d'exemple, admettons qu'un système possède 4 tâches t_1, t_2, t_3, t_4, utilisant toute la politique p. Cette politique est constituée de trois sous-politiques (min, C_1), (max, C_2) et (min, C_3). On a $C_1(t_1) = C_1(t_2) = 1$ et $C_1(t_3) = C_1(t_4) = 2$; $C_2(t_1) = C_2(t_2) = C_2(t_3) = 1$ et $C_2(t_4) = 2$; $C_3(t_1) = 1$ et $C_3(t_2) = C_3(t_3) = C_3(t_4) = 2$. Initialement partition $= \{\{t_1, t_2, t_3, t_4\}\}$, après la première passe, qui applique la sous-politique (min, C_1), on obtient : partition $= \{\{t_1, t_2\}, \{t_3, t_4\}\}$ et $o^1 = \{(t_3, t_1), (t_4, t_1), (t_3, t_2), (t_4, t_2)\}$. Après la deuxième passe (application de la sous-politique (max, C_2)), on obtient : partition $= \{\{t_1, t_2\}, \{t_3\}, \{t_4\}\}$ et $o^2 = o^1 \cup \{(t_3, t_4)\}$. Après la dernière passe (application de la sous-politique (min, C_3)), on obtient : partition $= \{\{t_1\}, \{t_2\}, \{t_3\}, \{t_4\}\}$ et $o = o^2 \cup \{(t_2, t_1)\}$. Au final, l'ordre est un ordre total dont la version simplifié est $o = 3 \prec 4 \prec 2 \prec 1$.

5.5.1 Sur les politiques dynamiques

Bien que les politiques d'ordonnancement puissent être définies à l'aide de variables représentant des caractéristiques dynamiques des tâches, accomplir la traduction de ces politiques de façon générale est très complexe. Comme les différences d'horloges (ou contraintes diagonales) sont exprimables grâce aux relations d'inhibition/permission, la possibilité de modéliser des ordonnanceurs comme EDF ou

FIFO est acquise. En effet, pour déterminer le niveau de priorité d'une tâche selon une de ces deux politiques, une seule horloge dynamique est nécessaire (rappel : EDF est définie par **min (D - d)** et FIFO par **max p**). Déterminer quelle est la tâche la plus prioritaire revient alors à comparer deux-à-deux ces niveaux de priorité, ce qui revient à comparer deux-à-deux les valeurs d'horloges, impliquant donc l'utilisation de différences d'horloges.

Seulement, ce raisonnement, bien qu'applicable dans les cas énoncés ci-dessus, n'est absolument pas généralisable. Alors qu'une politique est définie par une combinaison linéaire quelconque, il n'est pas possible, avec le raisonnement précédent, de traiter des horloges dynamiques avec un coefficient différent de 1. De plus, il suffit de rajouter une horloge à la donnée de la politique (comme Least Laxicity First : **min (D - d) - (C - c)**) et la traduction s'en trouve compromise : il n'est pas évident qu'il soit possible de traduire de telles politiques.

En dernier point, il faut pouvoir combiner dans une même politique, une sous-politique statique et une sous-politique dynamique. Une telle composition mixte n'est pas triviale.

Cette somme de réflexions a contribué à laisser de côté la prise en compte des politiques dynamique, afin de nous permettre de nous concentrer de manière plus efficace sur les autres aspects de la conception du langages Pola.

5.6 Traduction d'une allocation

Cet élément est central dans une description Pola, puisque c'est grâce à lui que les tâches vont pouvoir recevoir les ressources nécessaires à leur exécution. Le comportement d'une allocation est assez basique, puisqu'elle fait simplement passer les ressources de l'une à l'autre des tâches, ou permet à une tâche d'utiliser une ressource libre. Si l'on se contente de traduire l'élément allocation, sans prendre en compte son contexte, l'ensemble de transitions générées sera sous-optimal puisque de nombreuses transitions de cette allocation seront impossibles à tirer à cause de la politique d'ordonnancement. En effet, sans autre information, il est indispensable de générer *tous* les cas possibles d'échange des ressources entre tâches et cela, donc, sans distinction entre les tâches. Ce qui signifie que si deux tâches, t_1 et t_2, ont besoin de r pour s'exécuter, on doit regarder, pour chacune des tâches, si la ressource est libre ou si la ressource est occupée par l'autre tâche. Dans les deux cas, et en supposant que la ressource et les deux tâches sont préemptables, la tâche récupère alors la ressource. Mais si une politique vient contraindre le comportement et impose $t_2 \prec t_1$, alors t_2 n'a plus le droit de prendre directement la ressource à t_1. Ce qui signifie qu'une transition générée pour donner la sémantique de l'allocation n'est jamais tirable. Lorsque cela ne concerne qu'une seule transition, la gêne est bien sûr insignifiante.

Le problème réside dans le fait que le motif généré est hautement explosif, dans le sens où, pour un nombre réduit d'informations à la source (dans la spécification Pola), le motif peut être excessivement grand. Pour sensibiliser le lecteur à ce problème, nous allons calculer le nombre de cas possibles pour un exemple simple : celui d'une allocation allouant trois ressources à trois tâches.

Pour rappel, C_n^k signifie que l'on souhaite le nombre de possibilités de tirer k boules parmi n distingables. L'ordre d'apparition des k boules n'est pas important. On a également :

$$\mathrm{C}_n^k = \frac{n!}{k!(n-k)!} \tag{5.9}$$

Sans plus attendre, voici le nombre de transitions donnant la sémantique de l'allocation à n ressources et $nbtasks$ tâches :

$$nb = nbtasks * \left(\sum_{i=0}^{n-1} \left(\mathrm{C}_n^i \times \sum_{j=0}^{n-i} \left(\mathrm{C}_{n-i}^j \times (t-1)^{n-i-j} \right) \right) \right) \tag{5.10}$$

Cette équation calcule le nombre de transitions générées pour une tâche pour ensuite le multiplier par le nombre de tâches $nbtasks$ de l'allocation et ainsi obtenir le nombre total de transitions nb. Pour chaque tâche, i représente le nombre de ressources déjà utilisée par la tâche et j le nombre de ressources

libres. Par conséquent $n-i$ donne le nombre de ressources que la tâche ne possède pas et $n-i-j$ le nombre de ressources qui ne sont ni libres ni possédées par la tâche : elles sont donc possédées par les autres tâches. On calcule tout d'abord le nombre de possibilités de posséder i ressources parmi les n disponibles : C_n^i, on va ensuite multiplier ce nombre par la somme (pour avoir tous les cas de j) des nombres de possibilités d'avoir j ressources libres parmi les $n-i$ que la tâche ne possède pas (ce qui est donné par C_{n-i}^j), lui-même multiplié par les possibilités de placement des $n-i-j$ ressources, qui ne sont ni libres ni possédées par la tâche, sur les $t-1$ autres tâches (ce qui est donné par $(t-1)^{n-i-j}$).

Pour trois tâches et trois transitions, on a déjà un nombre conséquent de transitions puisque que c'est $63 * 3 = 189$ transitions qui sont générées rien que pour ce motif.

Toujours en restant dans cette idée de génération des allocations indépendantes du reste du système, une fois l'ensemble des transitions de toutes les allocations générées, il suffit de lier ces transitions par une relation de priorité correspondante à la politique d'ordonnancement. Ainsi, les transitions (des motifs allocation) des tâches sont prioritaires selon la politique d'ordonnancement.

Le caractère explosif du motif allocation peut-être grandement diminué si l'on considère la politique avant de générer l'ensemble des transitions. Ainsi, de nombreuses transitions sont éliminées car la politique interdit leur tir dans le contexte d'exécution sur lequel l'utilisateur n'intervient pas, c'est-à-dire sans comportements spécifiques. Une conséquence de la puissance d'expression des réseaux de Petri est que le comportement ajouté peut-être arbitrairement complexe. De ce point de vue là, les deux façons de traduire, en prenant en compte la politique ou non, donnent deux sémantiques différentes.

Avant de discuter plus avant de la différence de sémantique existant entre les deux approches et comme nous allons nous en servir comme argument, voyons le gain en terme de diminution du nombre de transitions générées que la méthode utilisant la politique d'ordonnancement permet.

Soit t tâches $t_1 \dots t_t$, avec $t_1 > \dots > t_t$.

$$nb = \sum_{a=1}^{t} \left(\sum_{i=0}^{n-1} \left(C_n^i \times \sum_{j=0}^{n-i} \left(C_{n-i}^j \times (t-a)^{n-i-j} \right) \right) \right) \tag{5.11}$$

Dans cette équation, on peut voir qu'au lieu d'appliquer la même formule à toutes les tâches, on va modifier progressivement le nombre de tâches desquelles on peut prendre une ressource. Plus a devient grand, plus le nombre de tâches moins prioritaires $(t-a)$ se réduit, et $(t-a)^{n-i-j}$, qui est un facteur d'explosion non négligeable, s'en trouve par la même occasion grandement réduit.

Pour comparaison avec notre exemple précédent de 3 tâches et 3 ressources, nous obtenons ici un motif comportant 96 transitions. Ce nombre reste important, mais il faut bien sentir que nous touchons au motif le plus sensible en terme de complexité et que nous avons là une diminution exponentielle de cette complexité (la complexité de la méthode, malgré une diminution exponentielle, n'en reste pas moins exponentielle).

5.6.1 Utiliser ou ne pas utiliser la politique d'ordonnancement, telle est la question

Si l'on ne prend pas en compte la politique pour la génération des motifs allocation, il suffit, d'une manière ou d'une autre, de désactiver une transition allocation prioritaire pour que son effet contraignant disparaisse, laissant libre champ à une transition ayant normalement une plus faible priorité. Au contraire, considérer la politique lors de la génération des motifs allocations ne permet pas ce comportement puisque les transitions à qui l'on a laissé le champ libre dans la version précédente n'existent tout simplement pas.

Les deux sémantiques ont des arguments aussi bien en leur faveur qu'en leur défaveur. La flexibilité et la puissance de contrôle de l'ordonnanceur sont à la faveur d'une génération "aveugle", mais du coup permettant des comportement terriblement tordus et complexes à corriger suite à une erreur de manipulation. Comme nous l'avons vu, la génération explosive du motif joue également en la défaveur de cette méthode. D'un autre côté le comportement est plus facilement prévisible si l'on choisit de prendre en compte la politique lors de la génération du motif, mais évidemment, des transitions activables sont perdues, d'où une perte de flexibilité. Perte qui va de pair avec une génération moins explosive du motif.

Un argument pour nous décisif contre une génération “aveugle” est que ce que l'on gagne en flexibilité, se fait au détriment de ce pourquoi nous avons conçu le langage. Avoir besoin des transitions supplémentaires, manipulables uniquement par le biais de l'utilisation de comportements (complexes) spécifiques, est un aveu de faiblesse du langage. Nous pensons préférable de garder l'esprit d'un DSL et donc corriger cette faiblesse en augmentant ou en modifiant le langage nous semble la méthode la plus digne d'intêret.

Ce langage sert à la fois à raccourcir considérablement le temps de modélisation d'un système temps réel, du point de vue de l'utilisateur, mais également permet à la communauté de la vérification de se rendre compte des parties critiques. En inspectant la traduction il est possible d'étudier les caractéristiques pouvant avoir un impact négatif sur l'efficacité (ou même la terminaison) de l'analyse ultérieure.

Nous avons donc choisi de prendre en compte la politique d'ordonnancement pour la traduction du motif allocation, et de laisser à des travaux futurs le soin d'étendre le langage en vue de lui ajouter d'éventuelles caractéristiques qui auraient peut être pu être spécifiées par un comportement spécifique si l'on avait fait l'autre choix.

5.6.2 La traduction

Maintenant que ce problème de sémantique a été discuté, voyons en détail la génération du motif d'une allocation a, donné par l'algorithme 11 et par la figure 5.10. Dans un premier temps, il nous faut rappeler et présenter des outils mathématiques dont nous allons avoir besoin pour manipuler les ensembles de tâches et de ressources de l'allocation.

Définition 28 - Ensemble des parties
L'ensemble des parties d'un ensemble s est noté $\mathcal{P}(s)$ et $\mathcal{P}(s) = \{i | i \subseteq s\}$

Définition 29 - Partition
Une partition d'un ensemble S est un ensemble P tel que
- $(\forall i \in P)(i \neq \emptyset)$
- $\bigcup_{i \in P} i = S$
- $(\forall (i,j) \in P^2)(i \neq j \Rightarrow i \cap j = \emptyset)$

Définition 30 - Répartition
La répartition d'un ensemble A sur un ensemble B est une application bijective $\mathcal{R} : B \rightarrow C$ telle que C est une partition de A.

Une répartition admet quelques propriétés qu'il est intéressant de connaître avant de passer à la suite.

Théorème 6 *Si $\mathcal{R} : B \rightarrow C$ est une répartition de A sur B, alors* $\text{Card}(A) \geqslant \text{Card}(B)$.

Preuve Imaginons que $\text{Card}(A) < \text{Card}(B)$. Pour que $\mathcal{R}$ soit une application bijective, il faut un même nombre d'élément pour B et C, d'où $\text{Card}(B) = \text{Card}(C)$. Comme $\text{Card}(A) < \text{Card}(C)$, alors
soit $(\exists i, j \in C^2)(\exists x \in A)(i \neq j \wedge x \in j \wedge x \in i)$
soit $\emptyset \in C$
Dans un cas comme dans l'autre, cela implique que C n'est pas une partition de A. C'est en contradiction avec la définition de répartition, donc l'hypothèse est fausse et ainsi $\text{Card}(A) \geqslant \text{Card}(B)$. □

Théorème 7 *Si $\mathcal{R} : B \rightarrow C$ est une répartition de A sur B, avec $A = B = \emptyset$, alors $\mathcal{R} = \emptyset$, c'est-à-dire l'application vide.*

Preuve Ce théorème est directement déduit de la définition. □

Algorithme 11 : Traduction d'un élément allocation

Données : une allocation a
un ordre $\prec$ issu de la politique d'ordonnancement
$Rnop = \{a.res \mid a.res.preempt = false\}$
$Tnop = \{a.tasks \mid a.tasks.preempt = false \vee \neg(x \prec t)\}$

```
1 pl alloc.active : alloc.active ( (a.noinit ?0 :1) )
2  (∀ t ∈ a.tasks )(
3  |  pl alloc.t.active : alloc.t.active ( (a.noinit ? 0 : 1) )
4  |  (∀ got ∈ P(a.res) \ a.res )(
5  |  |  (∀ free ∈ {i ∪ Rnop | i ∈ P(a.res \ got \ Rnop)} )(
6  |  |  |  (∀ othergot ∈ P(a.tasks \ Tnop \ {t}) )(
7  |  |  |  |  (∀ rep ∈ ℜ^{a.res\got\free}_{othergot} )(
8  |  |  |  |  |  tr a.t : a.t [0,0] a.active ?1 a.t.active ?1 a.t.released ?1
   |  |  |  |  |  (∀y ∈ got)(t.y?1)(∀k ∈ free)(SYS.k)
   |  |  |  |  |  (∀u ∈ othergot)((∀m ∈ rep(u))(u.m))
   |  |  |  |  |  (∀z ∈ a.res \ got)(t.z ?-1 -> t.z)
   |  |  |  |  )
   |  |  |  )
   |  |  )
   |  |  )
   |  )
   )
```

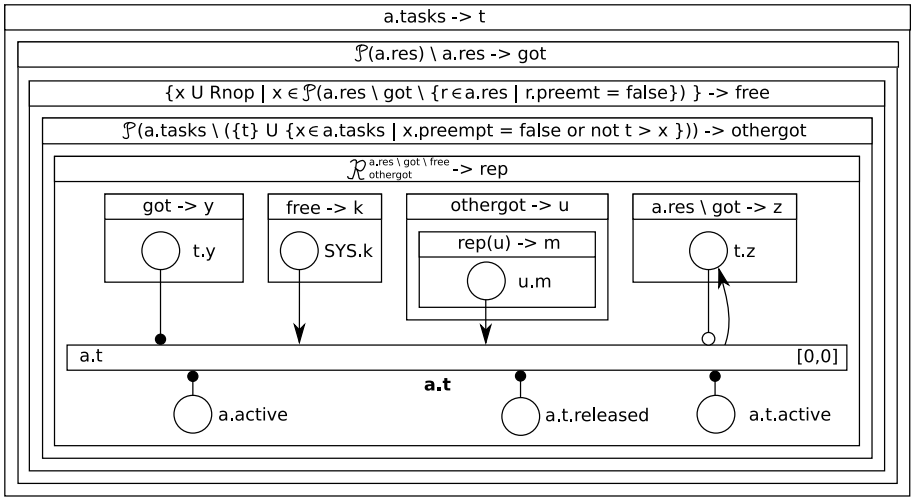


Fig. 5.10 – Traduction de l'élément allocation

Définition 31 - Ensemble des répartitions
L'ensemble des répartitions de A sur B est noté $\Re^A_B$.

L'allocation admet un état d'activité général, donné par la place *a.active*. Elle est également décomposée en autant de sous-allocations que de tâches déclarées l'utilisant. A chacune de ces sous-allocations on associe également un état d'activité *a.t.active*. Si une allocation est marquée **noinit** (ligne 1 de la sémantique) alors elle et ses sous-allocations sont inactives. Laissant ainsi l'opportunité d'activer individuellement une sous-allocation par l'utilisateur. Parce qu'il permet de spécifier des comportements complexes, ce mécanisme de désactivation d'allocation est à manipuler avec beaucoup de soin.

Pour donner la sémantique de l'élément allocation, il faut recenser tous les cas de possession de ressources par des tâches. Nous utiliserons pour cela les outils présentés plus haut. Comme nous l'avons déjà dit, chaque allocation est composée de sous-allocations. La sémantique de l'allocation est donc

définie par l'ensemble des sémantiques de ses sous-allocations, une sous-allocation étant donnée par toutes les combinaisons d'attribution des ressources à l'une des tâches $t \in a.tasks$ (ligne 2).

La tâche t peut déjà posséder une ou plusieurs des ressources nécessaires à son exécution. Il importe donc de recenser en premier lieu tous les cas de possession *a priori* des ressources par t. En considérant qu'un ensemble de ressource donne une combinaison de ressources possédées par t, l'ensemble $a.res$ représente le cas où t possède déjà toutes les ressources et pour lequel l'allocation est inutile. De plus, l'ensemble $\mathcal{P}(a.res)$ donne l'ensemble des parties de $a.res$ et donc l'ensemble des combinaisons de possession des ressources par t. L'ensemble des parties de $a.res$ recense donc tous les cas possibles, desquels on doit tout de même soustraire $a.res$ puisqu'il fait partie de $\mathcal{P}(a.res)$, d'où l'itération sur $\mathcal{P}(a.res) \setminus a.res$ de la ligne 4 de l'algorithme 11.

Une fois que l'on a l'ensemble des ressources qui sont déjà possédées par la tâche, il nous faut considérer toutes les possibilités pour une ressource d'être libre. Parmi les ressources que t ne possède pas déjà, il peut y en avoir qui sont libres et d'autres qui ne le sont pas. Parmi celles qui ne sont pas libres, on en trouve qui sont préemptables et d'autres qui ne le sont pas. Ces dernières ne sont pas récupérables par t : elles ne sont ni libres ni préemptables.

Pour allouer des ressources non préemptables il est donc impératif qu'elles soient libres. $Rnop = \{a.res \mid a.res.preempt = false\}$ donne l'ensemble des ressources non préemptables, par conséquent $X = a.res \setminus got \setminus Rnop$ donne l'ensemble des ressources qui peuvent être ou ne pas être libres sans que cela n'empêche l'allocation d'avoir lieu. Nous considérerons que $\mathcal{P}(X)$ donne toutes les combinaisons de ressources qui sont libres. Puisque les ressources de $Rnop$ qui ne sont pas déjà possédées par t doivent nécessairement être libres (dans le cas contraire, l'allocation ne peut avoir lieu), on a : $\{i \cup Rnop \mid i \in \mathcal{P}(X)\}$ donne l'ensemble des possibilités pour les ressources (non possédées par t) d'être libres (ligne 5).

La prochaine étape est de considérer les tâches qui peuvent posséder les ressources qui ne sont pas libres et que t peut préempter. La tâche t ne doit par pouvoir se préempter elle-même, ni aucune des tâches qui sont marquées **not preemptable**, ni aucune des tâches **preemptable** qui ne sont pas strictement moins prioritaires qu'elle. L'ensemble $Tnop = \{a.tasks \mid a.tasks.preempt = false \vee \neg(x \prec t)\}$ dénote l'ensemble des tâches qui ne sont pas préemptables par t. Ainsi, $\mathcal{P}(a.tasks \setminus Tnop \setminus \{t\})$ donne l'ensemble des possibilités qu'une ou plusieurs tâches préemptables possède les ressources manquantes (ni libres, ni possédées par t) (ligne 6).

Arrivé à ce point, nous avons connaissance de quasiment toutes les combinaisons nécessaires. Il nous manque cependant encore un dernier lien entre les tâches préemptables susceptibles de posséder les ressources manquantes et les ressources manquantes proprement dites. Ces ressources sont données par l'ensemble $a.res \setminus got \setminus free$ (i.e. l'ensemble des ressources de l'allocation, privé de l'ensemble des ressources possédée par t, privé des ressources libres). L'opérateur $\Re_{othergot}^{a.res \setminus got \setminus free}$ nous permet de faire le lien manquant en répartissant les ressources manquantes sur les tâches préemptables. En effet, $\Re_B^A$ donne l'ensemble des répartitions de A sur B, c'est-à-dire qu'à chaque élément de B est associé une partie de A. Donc, dans notre cas, à chaque tâche préemptable (les tâches de l'ensemble othergot) est associé une partie des ressources manquantes. Dans le cas où $Card(A) < Card(B)$, l'ensemble des répartitions est vide et aucune transition n'est générée. Par contre, si $A = B = \emptyset$, l'ensemble des répartitions est $\emptyset$ et une transition est effectivement crée. C'est le cas quand, parmi toutes les ressources manquantes, toutes sont libres *et* qu'aucune transition n'est préemptable (ce qui est normal puisqu'il n'y a rien a préempter, toutes les ressources nécessaires étant libres) (ligne 7).

Il nous reste alors à écrire les transitions selon les combinaisons calculées précédemment. Tout d'abord, chaque transition de l'allocation n'est sensibilisée que si l'allocation et active (**tr** a.t : a.t [0,0] a.active ?1 a.t.active ?1). Pour toute ressource déjà possédée, on n'y touche pas, mais on *teste* sa présence : $(\forall y \in got)(t.y?1)$. Pour toute ressource libre (SYS donne le nom du système dans lequel se trouve la tâche t et $SYS.res$ donne le nom de la ressource res dans le même système, mais c'est également le nom d'une place qui permet de savoir que la ressource est libre) , on la prend directement : $(\forall k \in free)(SYS.k)$, de même que les ressources possédées par des tâches préemptables :$(\forall u \in othergot)((\forall m \in rep(u))(u.m))$. Enfin, pour toute ressource non possédée par t, on s'assure que t ne la possède pas et on lui attribue (le retrait étant effectué par les préconditions $SYS.k$ et $u.m$).

5.7 Assemblage des éléments

La traduction des éléments étant fournie, il est maintenant nécessaire d'étudier la façon de les assembler pour donner la sémantique du système complet.

Nous n'avons pas encore eu l'occasion d'en parler, mais un système Pola peut être vu comme une interaction entre des observateurs et un système incontrôlable (ou plutôt que l'on ne souhaite pas contraindre par un quelconque contrôle). De ce point de vue là, la problématique de précédence des évènements systèmes/observation devient alors essentielle. Les contraintes d'inhibition sont utilisées pour contraindre un séquencement correct entre les évènements d'un système Pola.

Un autre problème que nous allons également aborder ici est celui de la gestion des intervalles pour les éléments tels que le générateur de période. En effet, l'indéterminisme potentiel soulève quelques problèmes qu'il nous faut traiter, avant la composition en vue de l'obtention du système complet.

Finalement, une fois ces points abordés nous conclurons le chapitre par la composition des divers éléments en un réseau de Petri prêt à analyser.

5.7.1 Séquencement et priorité des évènements

Lorsque deux évènements peuvent survenir au même instant, on est confronté à un indéterminisme : l'un ou l'autre peut être effectué de manière *indéterminée* (et indéterminable). Si dans beaucoup de cas cet indéterminisme est une caractéristique souhaitable de la spécification, il existe des cas où la résolution de cet indéterminisme fait partie intégrante de la spécification et ne pas prendre en compte le caractère complètement déterminé d'un enchaînement d'actions est une faute grave. Ceci peut paraître évident pour le cas de système discrets, mais dès que l'on rentre dans le cadre temporel dense, cet indéterminisme a plus souvent été considéré comme désirable que l'inverse. Bien souvent, un évènement pouvant intervenir de manière temporellement indéterminée est considéré comme incontrôlable. En règle générale cela reste vrai, mais concevoir un déterminisme temporel est possible et indispensable pour certains instants et certains évènements, c'est une des raisons qui ont poussé à introduire les deux relations d'inhibitions/permissions et c'est ce déterminisme que nous allons imposer pour les cas que nous allons présenter ici.

allocations

La sémantique que nous avons donnée de l'élément allocation ne concerne que l'élément lui-même : la prise en compte de la politique n'est appliquée qu'à l'élément. Ainsi, si deux allocations différentes concernent deux tâches dont l'une est moins prioritaire que l'autre, alors la sémantique ne stipule pas que l'allocation de la tâche la moins prioritaire est moins prioritaire que l'allocation de la tâche la plus prioritaire. C'est clairement une violation de la politique, il nous faut donc empêcher ces comportements par une relation d'inhibition entre les allocations, selon l'ordre général du système, comme calculé section 5.5. Pour cela, nous composerons le réseau suivant avec la sémantique des allocations.

Données : A l'ensemble des allocations du système;
$\succ$ la relation d'ordre, comme calculée section 5.5
$(\forall a_1 \in A)($
  $(\forall a_2 \in A \setminus a_1)($
    $(\forall t_1 \in a_1.tasks)($
      $(\forall t_2 \in a_2.tasks)(t_1 \succ t_2 \Rightarrow$
        tr t1 : $a_1.t_1$ ->
        tr t2 : $a_2.t_2$ ->
        inh t1 > t2))))

actions et échéances

Lorsqu'une tâche peut se terminer au même instant qu'un évènement de dépassement d'échéance, il n'y a pas de choix à laisser : puisque la tâche peut se terminer avant l'apparition de l'évènement échéance, cela signifie que la tâche n'a *pas encore* raté son échéance et donc que l'évènement de dépassement d'échéance doit être empêché. A moins de considérer l'instant de l'échéance comme un instant à ne pas atteindre, hypothèse qui n'est pas la nôtre.

Pour empêcher l'évènement de dépassement d'échéance, deux cas peuvent se présenter : soit la tâche n'est constituée que d'une action et alors c'est celle-ci qui devra contraindre l'apparition de l'évènement de dépassement d'échéance, soit elle en est composée de plusieurs et seules celles portant l'inscription **endoftask** devront empêcher cet évènement. Dans tous les cas, une simple priorité entre une transition d'action et une transition d'échéance n'est possible directement que si la fin de l'action est complètement déterminée (i.e. si son intervalle d'exécution est réduit à un point). Dans le cas contraire, voir la section 5.7.2.

Algorithme 12 : Contraintes entre éléments actions et observateur de dépassement d'échéance

Données : t la tâche portant l'action et l'échéance à relier;
$t.d$: la transition codant l'évènement d'échéance;
A : l'ensemble des transitions d'actions codant une fin de la tâche t;

1 **pr** $(\forall t.a \in A)(t.a\ t.a.tau) > t.d$

actions/période/offset et allocation

Toujours en se plaçant dans le cas où les éléments n'utilisent que des intervalles réduits à un point (le cas général est traité par la suite), un autre enchaînement à déterminiser est celui des transitions ayant un effet sur le réveil ou l'endormissement des tâches. Les motifs du générateur de période et de décalage du premier réveil sont directement concernés car leur nature intrinsèque est de réveiller des tâches. Les actions, si elles sont marquées **endoftask** peuvent endormir une tâche et restituer des ressources, ou bien si elles rendent uniquement des ressources avec **giveback**, sont également problématiques. De plus, une tâche, par ses actions et des comportements spécifiques, peut en réveiller une autre, ce qui va poser un problème comme nous allons le voir.

L'ordonnanceur, par le biais des transitions allocations, donne les ressources nécessaires aux tâches prioritaires *dès lors qu'elles sont réveillées* et que les ressources nécessaires sont *disponibles*. Ceci implique que selon l'ensemble des tâches réveillées, la décision de la (ou les) prochaines tâches à exécuter, peut être radicalement différente. Par conséquent, l'ordre d'apparition des évènements réveillant des tâches par rapport à ceux de l'allocation les utilisant est extrêmement important. Le fait de libérer des ressources peut également être important puisque tant que les ressources nécessaires ne sont pas libres une tâche reste en attente de l'exécution. Le fait d'endormir une tâche peut également être problématique si il est effectué dans un milieu concurrent, car endormir une tâche veut dire des ressources libérées et une tâche prioritaire en moins demandant les ressources.

Maintenant, que se passe-t-il si une transition d'allocation est tirée au même instant qu'un générateur de période, par exemple, mais *avant* de manière logique ? Prenons un exemple : une ressource r non préemptable est libre dans le système. Elle est nécessaire à l'exécution de la tâche t actuellement en attente de son exécution et pour l'instant possédant la priorité la plus haute du système. La tâche k, qui est sur le point d'être réveillée, est plus prioritaire que t et nécessite également de posséder la ressource r. Si l'allocation a lieu avant le réveil de k, alors r est donnée à t, ce qui implique qu'après son réveil, k se retrouvera bloquée par t. Dans le cas contraire, l'allocation donne la ressource à k.

Nous avons là un cas de figure où une tâche qui est peut-être extrêmement prioritaire et critique, est réveillée mais empêchée de s'exécuter car l'ordonnancement a mal joué son rôle d'observateur.

Pour palier ce problème, il convient donc de mettre une priorité entre les transitions des actions, période, offset et les transitions d'allocations, de la façon suivante :

Algorithme 13 : Contrainte entre élément réveilleurs de tâches et allocations

Données : L : ensembles des transitions d'allocations du système;
A : ensemble des actions du système;
P : ensemble des générateurs de période du système;
O : ensemble des décalages de réveil;
pr $(\forall x \in A)(\forall y \in P)(\forall z \in O)(x\ x.tau\ y\ y.tempo\ z\ z.tempo) > (\forall a \in L)(a)$

Toute autre transition générant un jeton de réveil devra être construite avec un soin extrême, car alors l'utilisateur s'avance sans filets et s'expose à concurrencer l'ordonnanceur.

5.7.2 Gestion de l'indéterminisme des actions / périodes / offsets

Utiliser une simple priorité entre deux transitions dont l'une possède un intervalle quelconque peut être contre-intuitif dans bien des cas. Et, en l'occurrence, le séquencement des évènements d'un système ne peut s'opérer de la même façon que précédemment si un intervalle des motifs n'est plus réduit à un point. Dans le cas contraire, cela reviendrait à contraindre l'intervalle au lieu de contraindre l'ordre d'exécution.

Nous allons définir une notion de mots temporisés différente de celle initialement introduite dans [AD94].

Définition 32 lettres, mots et langages temporisés
Soit Σ *un alphabet,*
une **lettre temporisée** *est un couple* $\Sigma \times \mathbb{R}$*,*
un **mot temporisé** *est la concaténation (noté .) de lettres temporisées, avec* $(x, a).(y, b) \Rightarrow b \geqslant a$*.* λ *est l'élément neutre de la concaténation.*
un **langage temporisé** *est un ensemble de mots temporisés.*

Nous allons définir la manière d'enchaîner les évènements en définissant le langage des enchaînements. Dans cette optique un évènement sera représenté par une lettre, une de ses apparitions par une lettre temporisée, un enchaînement par un mot temporisé et l'ensemble des enchaînements par un langage.

Soit un évènement e réveillant une tâche et f un évènement d'allocation. Nous avons déjà vu précédemment que l'évènement e ne doit jamais avoir lieu *après* f dans le même instant temporel. C'est-à-dire, sous forme de mots temporisés : $(e, 21).(f, 21)$ est un mot possible mais $(f, 21).(e, 21)$ ne l'est pas.

Dans cette section nous nous attachons à étudier le comportement indéterministe des évènements. C'est-à-dire que nous pouvons avoir (e, x) avec $x \in [i, j]$, autrement dit $(e, x) \in \mathcal{L} \Rightarrow x \in [i, j]$, où $\mathcal{L}$ est un langage. Les cas où les deux évènements f et e se suivent temporellement $(\forall x \in \mathbb{R})(\forall y \in \mathbb{R})(x < y \Rightarrow (f, x).(e, y) \in \mathcal{L} \vee (e, x).(f, y) \in \mathcal{L})$ ne nous intéressent pas car cette succession temporelle vient d'un indéterminisme qui est voulu.

Les entrelacements que nous voulons éviter interviennent au même instant temporel. Deux cas d'entrelacement des deux évènements sont possibles. Le premier, donné par l'ensemble des enchaînements d'évènements suivant : $(\forall x \in [i, j])((e, x).(f, x) \in \mathcal{L})$, est le cas non problématique pour lequel l'observation du système opérée par l'allocation s'effectue après l'évènement influençant l'ordonnanceur et par voie de conséquence les possibilités de tir des transitions allocations. Nous utilisons l'intervalle $[i, j]$ car c'est dans cet intervalle que peut survenir e. C'est donc uniquement dans cet intervalle que f et e peuvent être simultanés (au sens où ils interviennent au même instant temporel).

L'autre cas d'entrelacement, $(\forall x \in [i, j])((f, x).(e, x) \in \mathcal{L})$ est celui que nous souhaitons éviter. Si f intervient, cela signifie que tout évènement e doit intervenir plus tard. Nous n'empêchons nullement e, car il est toujours possible au même instant que f, mais seulement *avant* (logiquement, pas temporellement) f. Si f intervient cela signifie que tous les e ont eu leur chance, et qu'ils préfèrent attendre avant de survenir.

Le langage $\mathcal{L}$ complet donnant les entrelacement des deux évènements e et f est donné par :

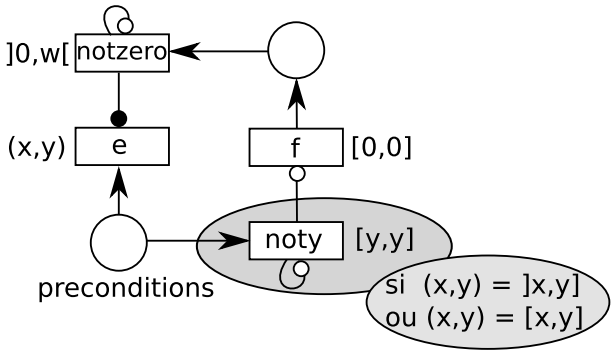
$$(\forall x \in (i,j))((e,x).(f,x) \in \mathcal{L} \wedge (f,x).(e,y) \in \mathcal{L} \Rightarrow y \in]x,j)) \tag{5.12}$$

L'intervalle noté (i,j) signifie que les bornes peuvent être quelconques. Ce qui exprimé ici est que l'on ne peut avoir les évènements enchaînés dans l'ordre $f.e$ que si e apparaît strictement (temporellement) après f. Une des conséquences les plus importantes est que, si la borne supérieure de (i,j) n'est pas stricte (c'est-à-dire $(i,j]$), il n'est pas possible à f d'intervenir à l'instant j car il n'est pas possible de trouver une valeur pour y qui soit à la fois dans (i,j) et strictement supérieure à j.

Rappelons le fonctionnement des contraintes d'inhibitions. Si e est une transition tirable dans $[x,y]$ et f une transition tirable dans $[0,0]$ (transition immédiate et urgente), alors $e > f$ empêche f de tirer mais contraint le tir de e car f ne peut dépasser sa borne maximale. Par conséquent e admet dans ce cas-là la même borne maximale que f. Dans le cas contraire $f > e$, il est obligatoire de tirer la transition f avant la transition e mais aucune des bornes des bornes des deux transitions n'est contrainte et e peut malgré tout tirer dans le même instant temporel (bien qu'obligatoirement *après* f).

Nous voyons donc qu'un mécanisme plus sophistiqué que les contraintes d'inhibitions seules est à mettre en place pour respecter de manière précise la sémantique souhaitée.

Il est possible d'avoir le comportement décrit un peu plus haut en utilisant le comportement suivant :



Lors du tir de la transition f, la transition e est rendue non tirable au même instant, car alors *notzero* devient sensibilisée, son horloge étant à zéro, elle contraint e, qui doit alors attendre une quantité de temps strictement supérieure à zéro pour pouvoir être tirée. Comme *notzero* n'est pas initialement sensibilisée, elle ne contraint pas la transition e qui peut tirer alors quand bon lui semble entre x et y, du moment que f n'est pas sensibilisée (si c'est le cas la borne maximale 0 empêche toute avancée du temps).

Dans le cas où e est tirable jusqu'à y inclus, alors, à cet instant-là, il faut empêcher f de tirer puisqu'un tir de f retarderait e, ce qui n'est pas possible. La transition *noty* est donc sensibilisé exactement comme e mais n'est pas tirable et contraint f lorsque son horloge atteint y.

Ce mécanisme introduit une nouvelle place qui, si elle n'est pas contrôlée, est non bornée. Ceci est bien évidemment inacceptable et des solutions existent permettant de ne pas rendre le réseau non borné. La solution la plus simple revient à un utiliser des flush-arcs, vidant la place en précondition. Si x admet en même temps un flush-arc venant de la place et génère un jeton, on est sûr que x ne générera pas plus d'un jeton dans la place.

Une autre solution serait de dédoubler la transition x pour tester si oui ou non la place est vide. Bien que parfaitement acceptable d'un point de vue théorique, ceci est inacceptable dans la pratique. En effet, les transitions x dans la pratique sont les transitions allocations. Or il a déjà été démontré que leur nombre pouvait être très important. Doubler ce nombre ne semble pas une alternative intéressante.

La solution adoptée est moins pure dans le sens où elle introduit un évènement artificiel, un effet de bord qui ne fait pas partie de la spécification Pola, servant uniquement au codage du comportement. Si le problème est que la place est non bornée, alors il suffit d'introduire une autre transition urgente

et immédiate, enlevant un jeton dès qu'il y en a deux dans la place. Ainsi la place ne pourra pas avoir plus de deux jetons.

La problématique de l'enchaînement des évènements d'un système peut ou non poser problèmes selon la logique que l'on souhaite utiliser. Par exemple, si l'on souhaite vérifier une formule de LTL, une différence dans l'enchaînement des actions ne va être discriminant que si cette différence entraîne des futurs différents. Ceci implique qu'il est théoriquement possible d'avoir des modèles plus simples, sans implémenter de manière stricte la sémantique des éléments Pola.

La même problématique s'applique entre une action terminant une tâche et un observateur d'échéance. Mais ce dernier n'agit pas sur le système, il est donc tout à fait inutile, pour toutes les logiques, d'implémenter une transition *notzero*. La transition *noty* reste, elle, utile.

5.7.3 Composition des éléments Pola

Il est temps de donner la sémantique de composition des éléments du langage Pola. En réalité elle est très simple, puisqu'elle se contente des deux opérations $\circ$ et $\odot$. La juxtaposition de réseaux $\circ$ est utilisée pour la composition des éléments déclaratifs, tandis que $\odot$ est utilisée pour composer un comportement au reste du système.

Chaque élément déclaratif est traduit dans un motif x qui sont regroupés dans un ensembles M. De même tous les comportements spécifiques sont stockés dans l'ensemble C. La sémantique de la spécification du modèle Pola est donné par la composition suivante, dont le résultat est le réseau de Petri P :

$$P = (\bigodot_{y \in C} y) \odot (\bigcirc_{x \in M} x) \tag{5.13}$$

Tous les noms de places et de transitions des comportements sont protégés (c'est-à-dire qu'il ne peut y avoir de conflits de noms entre les comportements et entre les motifs et les comportements), par conséquent seule la composition par étiquette peut rendre ces réseaux interdépendant du réseau généré donnant la sémantique du cœur déclaratif du langage.

La composition des motifs du cœur déclaratif est accomplie par la fusion des places qui servent d'interface aux motifs.

5.8 Conclusion

Nous avons donné dans ce chapitre la sémantique du langage Pola en terme de réseaux de Petri à inhibitions/permissions. Notre objectif initial était d'apprécier, en ce faisant, l'adéquation des *ipTPN* pour la modélisation du sous-ensemble des systèmes temps réel dont le périmètre est défini par Pola. Nous avons vu que le formalisme des *ipTPN* permet effectivement de donner une sémantique à Pola. Seule la sémantique des politiques dynamiques n'a pas été donnée, mais il a été remarqué que certaines de ces politiques dynamiques sont traduisibles, tandis que pour d'autres, cette traduction est très vraisemblablement impossible à donner.

Nous avons également vu que la sémantique d'un ordonnanceur en *ipTPN* est hautement explosif, ce qui correspond à la forte combinatoire de l'élément allocation. Dans cette traduction, la combinatoire est reportée dans le modèle, mais pourrait être sûrement déportée uniquement dans l'analyse en utilisant des données et des structures de contrôle plus flexibles pour spécifier cet ordonnanceur de manière plus concise.

Chapitre 6

L'outil Pola : Etudes de cas et expérimentations

La sémantique du langage Pola permet de donner une représentation du langage exprimée selon le formalisme des réseaux de Petri *ipTPN*. La traduction issue de la sémantique donnée dans le chapitre précédent, ainsi que la chaîne de vérification automatique a fait l'objet d'une implémentation que nous présentons dans ce chapitre. La vérification des *ipTPN* n'étant pas encore suffisamment mature, nous avons préféré utiliser une implémentation antérieure n'utilisant qu'une variante de la relation d'inhibition, la relation de priorité utilisée dans [BPV07]. La traduction de Pola dans ce formalisme n'est que partielle, car il souffre de problèmes de composition, mais permet néanmoins la vérification des cas d'études proposés. Le lecteur pourra oublier ce fait au cours de la lecture du chapitre, tout en gardant à l'esprit que les nombres issus de la vérification ne sont donc qu'indicatifs. En effet, la traduction utilisant uniquement la relation de priorité introduit des comportements passagers étrangers à la sémantique de Pola qui ne sont présents qu'à des fins de codage de la sémantique, grossissant ainsi la taille du graphe de comportement.

La chaîne de vérification de l'outil Pola est expérimentée sur des exemples reprenant les cas d'études étudiés dans le chapitre 4. Ces modèles sont déclinés en variations nous permettant d'exhiber des facteurs de complexité et de donner une estimation des limites de l'approche automatique.

6.1 Architecture

Le but d'un langage spécifique est de permettre à l'utilisateur d'utiliser un langage le plus proche sémantiquement des systèmes qu'il a à modéliser et à vérifier. Une attention toute particulière a été apportée pour que ce soit le seul langage qu'il ait à manipuler : il serait dommage qu'il soit obligé de manipuler d'autres formalismes, pas forcément aussi bien adaptés à son domaine, lors des étapes de la chaîne de vérification. Ceci est bien sûr une ligne de conduite idéale qu'il n'est pas toujours possible de tenir. Néanmoins ce but a conditionné la construction de notre architecture. Puisque seul le langage Pola est supposé connu, tout le reste du processus de vérification doit être au maximum déduit.

La première étape de la chaîne de vérification consiste en la génération d'un réseau de Petri temporel (avec les extensions nécessaires). Associé à ce réseau, une batterie de formule *LTL* est extraite du modèle Pola, selon les propriétés attendues dans le modèle. Le réseau est ensuite passé en entrée d'un explorateur d'états, permettant la construction d'un graphe d'états exhaustif, représentant l'ensemble des comportements du système et préservant la logique *LTL*. La validité des formules générées un peu plus tôt est alors vérifiée par un *model checker*. Le résultat de la vérification de l'ensemble des formules est alors transmis à l'utilisateur.

L'architecture que nous venons de présenter est résumé par le schéma de la figure 6.1.

Ceci donne l'aspect idéal du processus automatique. Mais cette automaticité n'est pas toujours possible. Deux raisons (et un troisième moins importante) à cela.

La première est une limite théorique. Comme nous l'avons déjà signalé, tout langage dont on veut

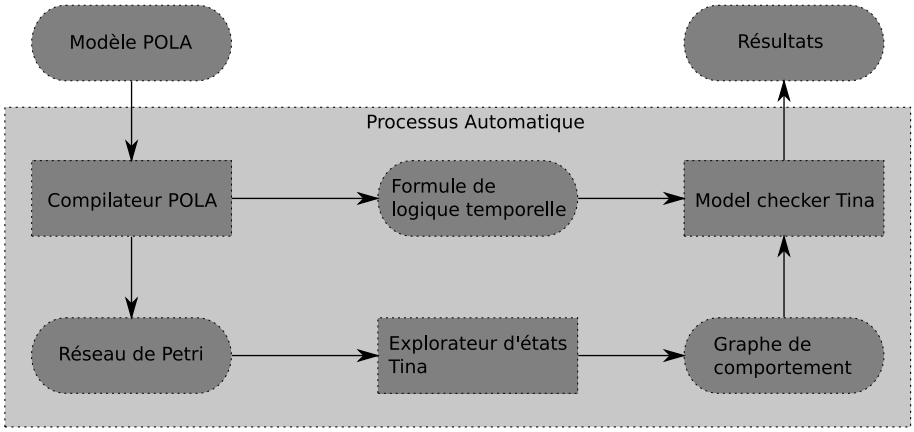


Fig. 6.1 – Architecture de l'outil Pola

exploiter la sémantique est soumis à une barrière théorique, appelée *indécidabilité* affirmant qu'au delà d'une certaine expressivité, il existe des questions pour lesquelles il n'est plus possible d'utiliser un unique procédé (un algorithme) y répondant à tous les coups (c'est-à-dire pour tout modèle que l'on a voulu soumettre à la question[1]). C'est le cas pour les réseaux de Petri temporels utilisant des chronomètres : il n'est pas possible d'avoir un algorithme (donc terminant au bout d'un temps fini) permettant de donner tous les états accessibles *à coup sûr*. Par contre, il est possible d'avoir un «algorithme» donnant une réponse (en temps fini) pour certains modèles. Le terme de *semi*-algorithme est préféré et c'est un tel *semi*-algorithme d'exploration que nous utilisons pour obtenir l'ensemble des comportements d'un réseau utilisant des arcs à chronomètre.

Il est donc certains modèles que notre processus ne pourra pas traiter pour une cause théorique, mais en plus de cela, il est un autre problème, pratique cette fois, empêchant le processus de donner une réponse. Lorsqu'on souhaite savoir si le système respecte telle ou telle propriété, il est présupposé que c'est pour tous les cas *possibles*. C'est-à-dire que certaines propriétés, comme "il n'y a jamais de panne dans le système", doivent s'assurer de l'absence de panne pour toutes les exécutions et/ou configurations possibles. Nous l'avons déjà vu, c'est le graphe qui représente le comportement du système, et c'est à partir de lui qu'une propriété est contrôlée. Cela signifie qu'il est primordial d'avoir l'ensemble du graphe pour obtenir une réponse correcte à une question du type absence de pannes. Or plus un système est complexe, plus son graphe de comportement est gros, sans que l'on puisse ni imposer, ni prévoir de borne maximale à la taille du graphe *complet*. Par contre, la mémoire dans laquelle est rangée ce graphe est, elle, bornée. D'où l'impossibilité pratique de stocker en mémoire certains comportement trop importants (soit par leur complexité, soit par leur longueurs, etc. Les causes étant très nombreuses). Ce phénomène est appelé *explosion combinatoire*, en raison du rapport de taille entre le modèle que l'on souhaite vérifier et son graphe de comportement.

Plutôt qu'un problème de taille mémoire, il existe une version temporelle de l'explosion combinatoire : l'ensemble des états étant si long à calculer que le coût de l'analyse devient prohibitif, même si le graphe peut effectivement tenir en mémoire.

Un troisième aspect, moins critique, est que la formule donnée en entrée du *model-checker* n'est pas fournie explicitement par l'utilisateur. En effet, comme nous l'avons déjà précisé, l'utilisateur n'a pas à avoir de connaissances sur le langage de formules utilisé et de plus nous n'avons pas de moyens, à l'heure actuelle, de spécifier à l'intérieur d'un modèle Pola les propriétés attendues. C'est pourquoi nous avons utilisé des propriétés très courantes, utiles dans beaucoup de situations. Pour prendre en compte d'autres propriétés, il faut donc ajouter *manuellement* les formules correspondantes. Mais, une fois cette étape accomplie, le processus redevient automatique.

[1] Aucun rapport avec le procédé moyen-âgeux du même nom : celui-ci respecte la convention de Genève.

6.1.1 Propriétés

Nous vérifions des propriétés dont la validité est souvent nécessaire pour qu'un système temps réel soit dit correct. La logique LTL que nous utilisons n'intègre pas directement de quantification temporelle à ses opérateurs. Mais ceci n'est pas absolument nécessaire car ces quantifications sont possibles en utilisant ou intégrant des observateurs du système (voir [BF99] ou plus récemment [GL06] pour des travaux utilisant cette technique et [MNP06] pour une traduction de formules *MITL* en automates temporisés).

Ordonnançabilité

La première propriété, l'ordonnançabilité du système, est vérifiée selon cette technique d'observateur. Ne pas utiliser d'observateurs signifie qu'il faut s'assurer par la logique que la terminaison de la tâche respecte son échéance, c'est-à-dire, en utilisant la logique *MITL* [AFH96] :

$$\Box(\text{réveil}_{T_i} \rightarrow \Diamond_{\leqslant \text{Echéance}} \text{Fin\_de\_tâche}_{T_i}) \tag{6.1}$$

ce qui veut dire que l'on a *toujours* ($\Box$), pour toutes les exécutions, la propriété qu'un évènement de réveil de la tâche T_i est *finalement* ($\Diamond$) suivi d'un évènement Fin_de_ tâche$_{T_i}$ de la même tâche. L'opérateur $\diamond$ porte une information de distance temporelle "$\leqslant$ Echéance", signifiant que l'évènement Fin_de_tâche$_{T_i}$ est au maximum distant de réveil$_{T_i}$ de Echéance unités de temps.

Cette formule permet donc de vérifier qu'aucune tâche ne rate son échéance. Mais au lieu d'utiliser un opérateur portant une quantification temporelle il est possible d'utiliser les éléments déjà présents dans le modèle ou d'en intégrer de spécifiques de manière à les utiliser comme des repères temporels lorsque la logique ne le permet pas. Ainsi, l'échéance d'une tâche est une partie intégrante du modèle de tâche et peut être utilisée comme repère temporel : si cet évènement survient, cela signifie que la formule ci-dessus est fausse. D'où la version LTL :

$$\Box\neg\text{Echéance\_ratée} \tag{6.2}$$

Cette formule signifie que l'échéance n'est jamais ratée : pour toutes les étapes d'une exécution, et pour toutes les exécutions, l'évènement *Echéance_ratée* n'apparaît pas.

Vivacité

Un système qui ne fait rien n'est pas couvert par la propriété ci-dessus : puisqu'il ne fait rien, il ne peut rater ses échéances. Pour s'assurer que le système n'est pas trivialement correct, il convient de vérifier qu'il travaille effectivement. Un système qui se bloque peut par exemple respecter la propriété d'ordonnançabilité ci-dessus, ce qui n'est évidemment pas acceptable, surtout pour les systèmes temps réel critiques.

Les vérifications de vivacité des tâches que nous opérons sont accomplies à l'aide de deux formules LTL. La première :

$$\Diamond\text{Fin\_de\_tâche}_{T_i} \tag{6.3}$$

vérifie que pour toutes les exécutions possibles, la tâche T_i est au moins exécutée une fois. En voici une lecture littérale : l'évènement Fin_de_tâche$_{T_i}$ arrive quelque part dans le futur de l'état initial.

Nous observons la fin de la tâche car c'est une garantie qu'elle a été effectivement terminée, plutôt que seulement démarrée. Mais il faut également faire remarquer que connaître l'évènement de démarrage d'une tâche est difficile. Il n'y a pas d'évènement explicite de démarrage d'une tâche : elle démarre dès que ses ressources sont allouées. Il est donc normalement possible de repérer le démarrage d'une tâche en observant les transitions de l'allocation correspondantes. Mais ceci ne garantit pas que nous allons observer *tous* les démarrages : il est possible qu'un comportement spécifique soit introduit qui fasse la même chose. Extraire automatiquement *toutes* les transitions permettant le démarrage d'une tâche n'est donc pas trivial.

Le lecteur attentif pourra justement se demander s'il n'est pas également possible d'utiliser uniquement la partie comportement afin de terminer une tâche. C'est effectivement le cas, mais nous supposerons que c'est un mauvais usage du langage Pola : tous les mécanismes sont présents pour que l'utilisateur précise les points de terminaison d'une tâche (au contraire de son démarrage) et ne pas utiliser ces mécanismes revient à ne pas utiliser *délibérement* une partie du langage Pola. Poussé à l'extrême, puisque la sémantique de Pola est donnée sous forme de réseau de Petri, il est tout à fait possible de se passer complètement de *toutes* les constructions Pola, et dans ce cas là notre outillage n'est évidemment plus adéquat !

La formule 6.3 ne reflète pas la vivacité d'une tâche dans sa totalité puisque seule *une* exécution de la tâche est nécessaire à la validation de la formule. Par contre, elle est un moyen de cibler un problème lorsque la vivacité, que nous allons voir, n'est pas assurée. La non validité inattendue de cette formule est souvent la marque d'un bug de la modélisation plutôt que la spécification elle-même.

L'autre formule concernant la vivacité d'une tâche qui est également vérifiée est la suivante :

$$\Box\Diamond \text{Fin\_de\_tâche}_{T_i} \tag{6.4}$$

On peut remarquer que pour construire cette nouvelle formule, nous avons utilisé la formule précédente, en la préfixant par $\Box$, ce qui signifie que $\Diamond \text{Fin\_de\_tâche}_{T_i}$ doit être valide tout le long de l'exécution (au lieu de l'état initial dans la version sans $\Box$) et cela pour toutes les exécutions. Cette formule vérifie donc que pour chaque exécution, et à chaque étape de ces exécutions, l'évènement $\text{Fin\_de\_tâche}_{T_i}$ interviendra dans le futur. Si la formule est vraie, le système, défini par un ensemble d'actions fini, est nécessairement cyclique et ne possède pas de blocage, et on sait que la tâche sera exécutée infiniment souvent.

La figure 6.2 montre un compte-rendu de ces analyses pour le modèle de base Osek qui sera repris dans la suite (voir figure 6.3). En premier lieu, on peut voir le compte-rendu de l'explorateur d'états Tina, qui sauvegarde le graphe de comportement sous la forme d'un fichier compressé (format spécifique *ktz*), réutilisé en entrée des *model checker*. L'analyse des propriétés se fait de manière globale en vérifiant que la conjonction de toutes les formules est valide, mais aussi de manière locale, en vérifiant les formules pour chaque tâche. Ainsi sur l'exemple, il est d'abord vérifié qu'aucune échéance n'est ratée de manière globale, puis une à une pour chacune des tâches. Ensuite il est vérifié que chaque action se termine (grain plus fin que la terminaison des tâches) puis que chacune des actions est vivante, c'est-à-dire qu'il n'existe pas d'exécution telle qu'une d'entre elles cesse d'être exécutée.

Cet outil n'étant qu'un prototype, nous n'avons pas voulu aller plus loin, car ajouter d'autres formules implique un délai supplémentaire pour la vérification. Dans l'optique d'une intégration future des propriétés au modèle, ou d'une configuration de propriétés à valider, des travaux d'identification des propriétés les plus spécifiques sont à entreprendre sur le modèle des motifs de vérification de [KC05] ou [GL06].

6.2 Présentation des modèles

Nous souhaitons montrer dans ce chapitre des exemples d'utilisation concrète de la chaîne de vérification Pola. Nous avons étudié au chapitre 4 comment certains mécanismes de Osek ou Arinc 653 pouvaient être modélisés à l'aide du langage Pola. Ce sont ces deux cas d'étude que nous reprendrons ici, en faisant varier certains éléments pour constater les répercussions sur le processus de validation.

6.2.1 Modèles utilisant des groupes Osek

Ce cas d'étude du modèle Osek va être utilisé et modifié de façon à pouvoir constater les effets des paramètres que sont la méta-période et le nombre de tâches, sur la taille du graphe de comportement, la mémoire utilisée et le temps pris pour l'analyse.

```
Generating ktz ...
# net noname, 29 places, 26 transitions                              #
# bounded, not live, possibly reversible                             #
# abstraction        count      props      psets       dead       live #
#      states        57754         29         78          0      57749 #
# transitions        58978         18         26          7         18 #

Begin properties checking ...
**************************
*Deadline Misses Checking*
**************************
Loading graph behavior ... DONE
Is there any deadline miss in the system ?  NO
Does task indus.T1 miss its deadline ?  NO
Does task indus.T2 miss its deadline ?  NO
Does task indus.T3 miss its deadline ?  NO
********************
*Liveness Checking*
********************
Loading graph behavior ... DONE
For all possible execution, will the system execute action indus.T1.act1 ? YES
For all possible execution, will the system execute action indus.T2.act1 ? YES
For all possible execution, will the system execute action indus.T3.act1 ? YES
For all possible execution, will the system execute action indus.T3.act2 ? YES
Loading graph behavior ... DONE
Is live indus.T1.act1 ?  - TRUE
Is live indus.T2.act1 ?  - TRUE
Is live indus.T3.act1 ?  - TRUE
Is live indus.T3.act2 ?  - TRUE
```

Fig. 6.2 – Sortie de l'outil Pola

6.2.2 Base

Ce modèle comprend trois tâches ordonnancées dans un ordre croissant de période (*rate monotonic*). La tâche T1 possédant la période la plus faible, c'est elle qui est prioritaire. Les valeurs des périodes des deux autres tâches sont des nombres premiers afin d'obtenir une méta-période importante. Le système de tâche est composé de deux groupes : le premier qui est aussi le plus prioritaire est composé de la seule tâche T1 tandis que le second est composé des deux autres tâches T2 et T3. La tâche T3 est décomposée en deux actions et est interruptible (par la tâche T1) après la première.

Les échéances des tâches ont été fixées au plus serré : elles déterminent les pires temps de réponse des tâches. Ces pires temps ne sont pas surprenants : ils sont donnés par la somme des temps d'exécution des tâches plus prioritaires et du temps d'exécution de la tâche dont on calcule le pire temps d'exécution. Dans le cas général, calculer un pire temps d'exécution de cette façon est pessimiste, mais dans le cas présent, puisque les périodes sont des nombres premiers entre eux, la dynamique de ces périodes fait que tous les cas possibles d'agencement temporel des instances de tâche vont se présenter.

6.2.3 Variation 1

La première variation est très légère puisque seule la période de la tâche T1 est changée en :

period 30 ⇒ **period** 60

```
system osek is
  res vproc is not preemptable
  res proc is preemptable
  not preemptable task T1 is
    action act1 in [5,5] with alloc1
    period [30,30]
    deadline 5
    policy RM
  end
  task T2 is
    action act1 in [4,4] with alloc2
    period [73,73]
    offset [7,7]
    deadline 16
    policy RM
  end
  task T3 is
    action act1 in [7,7] with alloc2 giveback
    action act2 in [8,8] with alloc2 endoftask
    period [97,97]
    deadline 24
    policy RM
    behavior is
      tr a1end s1 -> s2
      tr a2end s2 -> s1
      pl s1 (1)
      lb indus.T3.act1 a1end
      lb indus.T3.act2 a2end
  end
  policy RM is min P
  allocation alloc1 is
    resources proc
    tasks T1
  allocation alloc2 is
    resources proc, vproc
    tasks T2, T3
end
```

Fig. 6.3 – Modèle de démonstration Osek

Doubler la période revient à doubler la méta-période dans ce cas, permettant ainsi de constater l'impact de cet allongement sur le nombre de classes du graphe de comportement.

Variations de la variation 1

Afin d'éprouver plus finement l'impact de la méta-période sur la complexité du comportement (i.e. la taille du graphe de comportement), nous pouvons faire varier la méta-période par un multiplicateur à chaque fois plus grand.

- Variation $1'$: la méta-période de la version de base est triplée par un triplement de la période de la tâche T1 à 90. Afin de conserver l'ordre de priorité, les périodes de T1 et T2 sont échangées : la période de T1 est maintenant de 73, tandis que celle de T2 est de 90.
- Variation $1''$: la méta-période de la version de base est quadruplée par un quadruplement de la période de la tâche T1 à 120. Afin de conserver l'ordre de priorité, les périodes sont décalées ainsi : la période de T1 est l'ancienne de T2 : 73, la période de T2 est l'ancienne de T3 : 97 et la période de T3 est la période T1 quadruplée : 120.

Une autre variation (variation $1'''$) est une expérimentation donnant une méta-période très proche de celle de la variation $1''$, mais obtenue avec de fortes différences entre les périodes : T1 possède une période de 7, T2 une période de 43 et T3 une période de 2819.

6.2.4 Variation 2

Cette variation est la même que la version de base à laquelle on a ajouté une nouvelle tâche : T4.

```
task T4 is
  action a1 in [1,1] with alloc1
  period [20,20]
  deadline 1
end
```

Les échéances sont augmentées de 1 pour T1 (6) et T2 (17) et de 2 pour T3 (26) pour toujours refléter les pires temps d'exécution (puisque T4 ne met qu'une unité de temps pour se terminer). Cette

nouvelle tâche fait partie du groupe 1, le plus prioritaire, visible par son utilisation de l'allocation *alloc1*.

6.2.5 Variation 3

Nous reprenons la variation 3 en ajoutant une nouvelle tâche T5. Cette tâche est la nouvelle tâche la plus prioritaire (et appartient au premier groupe). Sa période est égale à 15, sa capacité et son échéance sont égales à 1.

```
task T5 is
  action a1 in [1,1] with alloc1
  period [15,15]
  deadline 1
end
allocation alloc1 is
  resources proc
  tasks T1, T4, T5
```

Les échéances des autres tâches sont ainsi augmentées de 1 pour T1 (7), T2 (18), T4 (2) et de 2 pour T3 (28).

6.2.6 Variation 4, 4′ et 4″

Une nouvelle tâche est ajoutée, T6. Elle est soit la plus prioritaire du système dans la variation 4, soit la moins prioritaire dans les variation 4′ et 4″. Dans la version pour laquelle T6 est la plus prioritaire, elle admet une période de 10 unités et dans celles où elle est moins prioritaire, soit 146 unités de temps, soit 30 unités de temps. Son temps d'exécution est d'une unité de temps.

6.3 Influences des paramètres

Le tableau ci-après récapitule les résultats obtenus. Les colonnes *nb pl.* et *nb tr.* donnent respectivement le nombre de places et le nombre de transitions du réseau généré ; *nb états* et *nb arcs* donnent la taille du graphe respectivement en nombre de nœuds et en nombre de transitions (c.-à-d. la taille de la relation de transition) ; *temps* donne le temps total de l'analyse (compilation en réseau de Petri + calcul du comportement + *model checking*) ; *méta-période* donne le temps nécessaire pour un système d'accomplir une révolution de son comportement.

La colonne *mémoire* donne une approximation de la mémoire utilisée par la construction du comportement. L'analyse de cette occupation mémoire est rendue complexe à cause de l'utilisation de plusieurs langages dans l'outil Tina. Tina est implémenté en Standard ML et compilé par Mlton. Les modèles que nous utilisons nécessitent la gestion des arcs à chronomètres, ce qui implique que les classes sont stockées dans des polyèdres. La partie liée au calcul des classes pour ces modèles est donc faite en C car la bibliothèque de manipulation de polyèdre que Tina utilise est codée en C.

Mlton permet d'obtenir certaines informations sur le fonctionnement de son ramasse-miètes (*garbage collector*) et notamment l'information de la taille maximale des données utiles lors des ramassages. En indiquant d'effectuer un ramassage à la fin de l'analyse, lorsque le graphe est à sa taille maximale, nous obtenons l'information la plus fiable possible.

La mémoire de la partie en C n'est pas traitée par le ramasse-miètes de Mlton. Nous avons instrumenté le code de la génération du graphe de façon à connaître l'espace mémoire occupé par l'ensemble des classes. Cette occupation n'est donc pas l'occupation maximale, mais l'occupation utile : c'est l'espace pris par l'ensemble de toutes les classes mises sous forme canoniques.

Le résultat de l'occupation mémoire du calcul du graphe de comportement est donné par une addition indiquant en premier opérande la mémoire utilisée par la partie «SML» et en deuxième opérande la mémoire utilisée par la partie «C». La première information donne une indication de l'occupation mémoire de la structure du graphe (relation de transition) et la seconde donne une indication de l'occupation mémoire de l'ensemble des classes.

	nb pl.	nb tr.	nb états	nb arcs	temps	méta-période	mémoire (Mo)
OSEK base	29	26	57.754	58.978	20 s	212.430	10+36,1
OSEK v1	29	26	84.934	86.368	28 s	424.860	14,6+52,3
OSEK v1′	29	26	112.335	113.979	37 s	637.290	19,3+68,6
OSEK v1″	29	26	134.592	136.400	44 s	849.720	23+81,3
OSEK v1‴	29	26	632.868	674.912	3m53 s	848.519	111+415
OSEK v2	35	33	214.784	228.844	1m. 42s.	424.860	41,9+182
OSEK v3	42	45	368.098	428.736	4m. 27s.	424.860	81,6+425
OSEK v4	47	50	650.787	880.529	11m.30s.	424.860	165+1035
OSEK v4′	48	55	384.304	448.306	5m. 42s.	424.860	92+530
OSEK v4″	48	55	504.892	698.836	9m.40s.	424.860	129+825

Ce cas d'étude permet de constater que la méta-période est un facteur de complexité. Doubler, tripler ou quadrupler la taille de la méta-période entre les versions base et v1, v1′ et v1″ engendre une augmentation (linéaire) sensible du graphe de classe. L'augmentation est linéaire car les valeurs utilisées pour accomplir l'augmentation de la taille de la méta-période sont également augmentées de manière linéaire. Mais la complexité ne vient pas de la méta-période seule : la taille du graphe dépend aussi du pas temporel minimal (Pour une autre analyse de l'impact de la méta-période *seule*, voir le cas d'étude suivant). Ainsi, plus les différences entre les quantités temporelles utilisées sont grandes plus il y a de pas temporels. Ce facteur de complexité peut être constaté dans la version v1‴ pour laquelle les valeurs de période sont très hétérogènes et augmentent ainsi considérablement la taille du graphe de classes.

La méta-période n'est pas le seul facteur de complexité : le nombre de tâches du système en est un autre et c'est peut-être le plus important. Augmenter le nombre de tâches revient généralement à augmenter la combinatoire du système. C'est donc un facteur causant une augmentation exponentielle de la complexité. Cette progression exponentielle est observable dans les versions v2, v3 et v4 pour lesquelles la méta-période est strictement identique à la variation v1.

Cette augmentation peut ne pas être exponentielle si la tâche ajoutée possède une priorité trop faible. En effet comme nous l'avons vu dans le chapitre précédent, la sémantique des éléments allocations est très dépendante de la priorité : plus une tâche possède une priorité élevée, plus ses possibilités de préempter une autre tâche sont élevées. La variante v4′ montre cela : l'ajout d'une tâche possédant la plus faible priorité ne rajoute que peu de complexité au comportement en regard de celle apportée par des tâches plus prioritaires. Les deux variantes v4 et v4′ montrent donc les deux extrêmes en terme d'augmentation de la taille du graphe lorsque l'on ajoute une tâche au système. Par contre, la petitesse du comportement de la version v4′ n'est pas uniquement dû au fait que la tâche ajoutée est de faible priorité : la variation v4″ voit sa période diminuée par rapport à v4′ afin qu'elle soit proche de celle de la variation v4. On constate alors que la taille du comportement se rapproche de celle de la variation v4 : c'est un effet de la diminution du pas temporel. On ne peut conserver exactement la même période que celle de la variation v4, car le fait qu'elle soit la tâche la moins prioritaire implique que le système n'est alors plus ordonnançable.

6.4 Cas d'étude : Arinc 653

Ce cas d'étude permet de constater l'influence du pas temporel en relation avec la méta-période, ainsi que l'indéterminisme et le nombre de tâches sur la complexité de l'analyse.

6.4.1 Base

Le modèle est assez basique, il est composé d'un système de contrôle et de deux partitions décrites par deux systèmes. Le contrôleur limite le temps de l'une et de l'autre des partitions en les activant/désactivant. Les deux systèmes sont déclarés préemptables : les horloges des systèmes sont donc suspendues (au lieu d'être remises à zéro) lors d'une désactivation. Les deux partitions sont constituées

```
system arinc653 is
  behavior is
    tr topartition2 [50,50] p1 -> p2
    tr topartition1 [50,50] p2 -> p1
    pl p1 (1)
    lb partition1.active p1
    lb partition2.active p2
end
preemptable system partition1 is
  res proc is preemptable
  task T1 is
    action act1 in [10,10] with alloc
    period [50,50]
    deadline 50
    policy RM
  end
  task T2 is
    action act1 in [20,20] with alloc
    period [50,50]
    deadline 50
    policy RM
  end
  policy RM is min C
  allocation alloc is
    resources proc
    tasks T1, T2
  end
```

```
noinit preemptable system partition2 is
  res proc is preemptable
  task T1 is
    action act1 in [10,10] with alloc
    period [50,50]
    deadline 50
    policy RM
  end
  task T2 is
    action act1 in [20,20] with alloc
    period [50,50]
    deadline 50
    policy RM
  end
  policy RM is min C
  allocation alloc is
    resources proc
    tasks T1, T2
  end
```

Fig. 6.4 – Modèle de démonstration Arinc 653

de deux tâches, ordonnancées selon leur capacité : la tâche nécessitant le moins de calcul est effectuée en premier. Les périodes des tâches correspondent à la période des partitions.

6.4.2 Variation 1

Nous voulons montrer ici l'influence du pas temporel. Nous multiplions par 10.000 toutes les valeurs de la variation de base. La méta-période est ainsi également augmentée de 10.000 unités.

6.4.3 Variation 2

Cette variante modifie la précédente en divisant toutes les quantités temporelles liées à la partition 1 par 100.000. La partition 1 dure donc 5 unités de temps. La partition 2 est laissée telle quelle (sa durée est donc de 500.000).

6.4.4 Les variations de la variation 3

Ce système comprend plus de tâches et de partitions que dans les systèmes précédents, puisque c'est maintenant 4 partitions de 4 tâches chacune que nous allons étudier. Les quatre partitions durent respectivement 50, 100, 160 et 410 unités de temps. Soit C_j la durée de la partition j, la tâche T_i^j avec $i, j \in 1, 2, 3, 4$ s'exécutant sur la partition j, est définie par le triplet (capacité, période,échéance). D'où :

$T_i^1 = (i, C_1, C_1)$
$T_i^2 = (10 + i, C_2, C_2)$
$T_i^3 = (30 + i, C_3, C_3)$

$T_i^4 = (50 + i, C_4, C_4)$

Les périodes des tâches étant égales à celles de la partition à laquelle elles appartiennent, la méta-période est donnée par la somme des durées des partitions, soit 720.

La politique utilisée par toutes les tâches est **min** C.

Pour définir les variations, nous avons besoin de définir les comportements suivants :

ordre sur les périodes : les réveils simultanés des nouvelles instances de tâches apparaissent uniquement dans l'ordre de priorité définie par la politique.

périodes avant partitions : les changements de partitions sont effectués après l'apparition des réveils des nouvelles instances de tâches.

partitions avant périodes : les changements de partitions sont effectués avant l'apparition des réveils des nouvelles instances de tâches.

Les variations sur cette base seront les suivantes :

variation $v3'$: **périodes avant partitions**
variation $v3''$: **périodes avant partitions** et **ordre sur les périodes**
variation $v3'''$: **partitions avant périodes** et **ordre sur les périodes**

6.4.5 Résultats

	nb pl.	nb tr.	nb états	nb arcs	temps	méta-période	mémoire
Arinc 653	31	24	129	295	< 1 s	100	0,1 + 0,2
Arinc 653 v1	31	24	129	295	< 1 s	1.000.000	0,1 + 0,2
Arinc 653 v2	31	24	95	156	< 1 s	500.000	0,1 + 0,1
Arinc 653 v3	109	108	864.689	1.847.289	1h12m55s	720	1401 + 10240
Arinc 653 v3$'$	109	108	929	1.505	16s	720	1,5 + 18,2
Arinc 653 v3$''$	109	108	577	609	9s	720	1 + 11,3
Arinc 653 v3$'''$	109	108	107	107	1s	720	0,2 + 0,9

La version 1 montre que l'impact seul de la méta-période est négligable : les valeurs temporelles deux versions *v1* et *base* varient seulement d'un multiplicateur. Cela signifie que le système *v1* se comporte *exactement* de la même façon mais est seulement plus lent. L'abstraction regroupant les états ayant un comportement similaire sous forme de classes n'est pas perturbée par l'élongation temporelle des délais.

Nous avons déjà montré que le pas temporel pouvait avoir un impact sur la complexité du comportement. Pour obtenir un pas faible par rapport aux grandeurs du système, nous avions dit qu'il était nécessaire que les valeurs du système ne soient pas toutes homogènes. Ceci n'est pas suffisant, et nous le voyons bien sur la variation v2 : celle-ci est hétérogène mais comme les grandeurs hétérogènes sont isolées, le système se comporte comme une juxtaposition des deux systèmes indépendants. La complexité obtenue est donc l'addition des complexités des deux partitions.

La troisième variation est intéressante de par sa simplicité : le comportement d'un tel système est aisément appréhendable, mais certains détails viennent complexifier son analyse de façon déroutante. Le système admet une méta-période de 720, qui est la somme des durées des partitions. L'ensemble des tâches se termine avant la fin de la partition et la période est synchronisée sur la durée de la partition. Ainsi, à chaque exécution de la partition une nouvelle instance pour chacune des quatre tâches est réveillée. Un tel fonctionnement n'a rien de bien compliqué. Il est dès lors bien étonnant de constater que l'espace d'états d'un tel système est si imposant.

Cette complexité vient d'un indéterminisme qui en apparence n'est pas important, mais en pratique se révèle être d'une importance capitale. En effet, toutes les tâches possèdent la même période et la partition dure le temps des périodes des tâches. Nous avons donc 5 évènements intervenant de manière simultanée mais dont les différents enchaînements vont donner des états différents. De plus, parmi ces évènements, le changement de partition préempte la partition pour laquelle ces évènements de période apparaissent. Les états de la suite de l'exécution après préemption gardent en mémoire l'état de la

partition avant la préemption, et par conséquent les états donnant l'exécution d'une partition x seront différents selon l'état dans lequel la partition $x-1$ a été préemptée.

Plus précisément, les 4 transitions repérant le réveil d'une nouvelle instance, peuvent conduire à $\Sigma_{k=1}^{4} \frac{4!}{k!(n-k)!} = 15$ états différents. Ce nombre est la somme des combinaisons de marquages possibles après le tir de chacune des transitions de période. Mais ceci, en soit, n'a pas une influence importante puisqu'à la fin de la séquence il n'est possible de n'avoir qu'un seul état. On a donc une génération d'un paquet d'états amenant à un unique état "final". De cet état, il est encore possible de faire l'allocation de la tâche T1, puisque c'est toujours faisable en 0 unités de temps.

Là où les choses rendent la vérification franchement difficile, c'est qu'à partir de chacun des états générés ci-dessus, il est possible de préempter la partition. Chaque préemption entraîne que chaque état généré par la suite dépend de l'état de préemption d'origine. Ainsi, au lieu de n'avoir qu'un paquet de transitions dû à la combinatoire des apparitions des transitions de périodes, on se retrouve avec autant de sous-comportements que d'états où la préemption est possible. Et dans ces sous-comportements, le même raisonnement est également possible. D'où la taille disproportionné du graphe de comportement constaté pour la version v3.

La variation $v3'$ oblige la préemption à n'intervenir qu'après le tir des transitions de périodes. Ceci a pour conséquence la réduction dramatique de l'espace d'états puisque dès lors, la préemption n'est plus possible qu'à partir d'un nombre d'états très réduit en comparaison. La version $v3''$ s'occupe de la combinatoire des évènements de réveil en imposant une seule possibilité parmi les nombreuses possibles.

Même après la version $v3''$, il reste des cas d'indéterminisme non résolus : il est encore possible de choisir de faire l'allocation de T1 avant le changement de partition ou bien de changer directement de partition. Si l'on impose que le changement de partition est prioritaire ce comportement n'est plus possible. La version $v3'''$ est la plus simple possible : son comportement n'est qu'un chemin (cyclique), il n'y a pas de branchements (car il y a autant d'états que de transitions).

Nous voyons ici une application de l'utilisation de la relation d'inhibition pour imposer un ordre d'exécution, ce qui est assimilable à la déclaration d'une symétrie. Une abstraction utilisant une symétrie n'est pas bisimilaire avec le graphe concret, car des pans entiers d'exécutions possibles sont supprimés. Mais l'intérêt d'une abstraction doit toujours être étudiée selon les propriétés que l'on souhaite préserver. Et en effet, dans notre cas, il n'est pas important de connaître l'ordre dans lequel les évènements de période et de changements de partitions s'enchaînent. D'où la possibilité d'utiliser une telle symétrie. De plus, la symétrie est aisément démontrable puisque les partitions sont indépendantes. Mais ceci n'est par toujours le cas et trouver (et surtout prouver) une symétrie n'est généralement pas facile.

Il reste à faire une remarque concernant la grande différence de temps d'analyse entre des modèles Osek et Arinc 653 dont l'espace d'états est de taille comparable. Tout d'abord il est bien évident que la taille de la relation de transition est un facteur de lenteur : plus il y a de transitions sortant d'un état, plus il y a de nouveaux états dont il faut tester l'égalité avec ceux déjà visités. Mais il est un autre paramètre de lenteur pour le modèle Arinc 653 v3 : le nombre de tâches. En effet, puisque ce système est constitué de beaucoup de tâches, cela signifie qu'à chaque instant, un nombre conséquent de transitions sont sensibilisées. Ceci implique que le polyèdre représentant l'espace d'état d'une classe possède un grand nombre de dimensions et donc la complexité de la comparaison et des calculs de classes suivantes est du même coup accrue.

6.5 Corrections d'erreurs et raffinages quantitatifs

L'un des intérêts de la technique du *model checking* est de pouvoir fournir automatiquement, lorsqu'une propriété n'est pas vérifiée, une séquence démontrant son invalidité. Cette séquence est appelée séquence *contre-exemple*. Une telle séquence est alors utilisable pour corriger les erreurs du modèle, ou, le cas échéant, de la propriété.

6.5.1 Une erreur dans le modèle Osek

Imaginons qu'au lieu d'écrire :

tr a1end s1 -> s2
tr a2end s2 -> s1
pl s1 (1)

pour le comportement de la tâche T3 de la variante 1 du modèle Osek présentée dans ce chapitre, nous ayons spécifié :

tr a1end s1 -> s2
tr a2end s2 ->
pl s1 (1)

Ce dernier comportement ne réinitialise pas le comportement de la tâche T3, qui n'est dès lors plus exécutable. La tâche T3 ne se trouve plus ni dans l'état s_1, ni dans l'état s_2.

Plus grave : la tâche T3 n'est pas exécutable après sa première exécution, mais l'ordonnanceur ne possède pas cette information et donne malgré tout les ressources à la tâche T3. La tâche T1 peut préempter T3 mais T2 ne le peut pas et par conséquent son échéance est obligatoirement ratée.

Finalement, ni T3 ni T2 ne sont exécutables mais leurs générateurs de réveils périodiques sont, eux, toujours actifs. Ceci a pour conséquence de rendre le réseau non borné. C'est un problème connu, qu'il serait préférable de résoudre de façon primitive par l'emploi d'une construction Pola, mais qui doit pour l'instant être résolu à l'aide d'un comportement spécifique.

Borner un modèle Pola

Le problème vient de ce qu'il y a plusieurs façons de borner le comportement et que le sujet, que nous n'avons abordé que superficiellement dans le cadre de cette thèse, mérite une étude approfondie. Par exemple, dans le cas où l'on considère qu'un modèle n'a pas le droit de rater une échéance, il est possible d'arrêter complètement le système à la moindre échéance ratée. Il est également possible de borner le nombre de réveils en interrompant définitivement l'instance en cours. Comme la première méthode est plus radicale, elle nous fournit des résultats plus courts et est donc préférable dans le cadre de ce chapitre.

behavior is
tr t1 r a ->
lb indus.T2.deadline t1
lb indus.T2.released r
lb indus.active a

Comportement ajouté à T2

tr t1 r a ->
lb indus.T3.deadline t1
lb indus.T3.released r
lb indus.active a

Comportement ajouté à celui de T3

Le résultat donné par la chaîne de vérification est attendu :

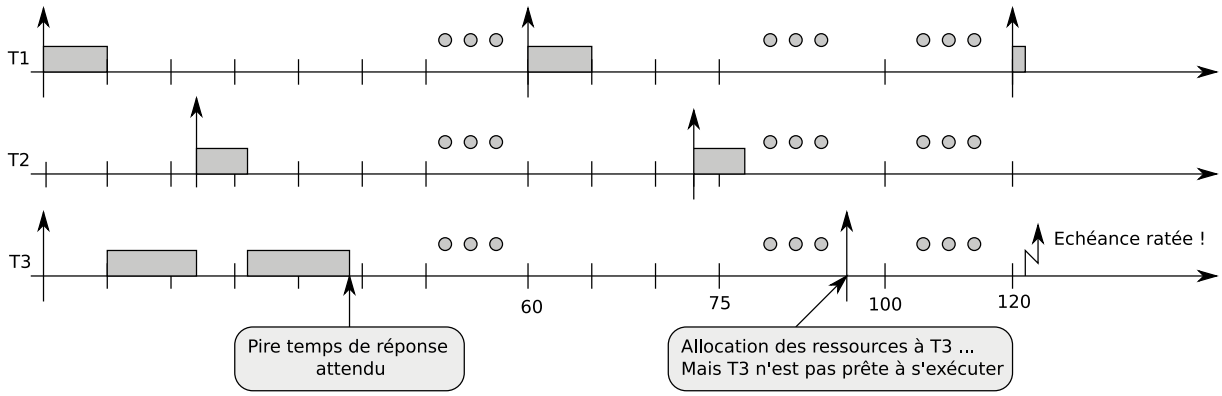
```
...
Is there any deadline miss in the system ?  YES
...
Does task indus.T2 miss its deadline ?  NO
Does task indus.T3 miss its deadline ?  YES
...
For all possible execution, will the system execute action indus.T2.act1 ?  YES
For all possible execution, will the system execute action indus.T3.act1 ?  YES
For all possible execution, will the system execute action indus.T3.act2 ?  YES
...
Is live indus.T3.act1 ?  - FALSE
Is live indus.T3.act2 ?  - FALSE
```

Les tâches T2 et T3 ratent leur échéance ; elles sont exécutées au moins une fois, mais ne le sont pas infiniment souvent (*not live*), ce qui indique qu'à partir d'un moment elles ne sont plus exécutées.

Le contre-exemple donné par le *model checker* est en fait équivalent au comportement du système :

$$s_0 \xrightarrow{indus.alloc1.T1} s_1 \xrightarrow{indus.T1.act1} s_2 \xrightarrow{indus.alloc2.T3} s_3 \xrightarrow{indus.T2.offset} s_4 \xrightarrow{indus.T3.act1} s_5 \xrightarrow{indus.alloc2.T2}$$
$$s_6 \xrightarrow{indus.T2.act1} s_7 \xrightarrow{indus.alloc2.T3} s_8 \xrightarrow{indus.T3.act2} s_9 \xrightarrow{indus.T1.period} s_{10} \xrightarrow{indus.alloc1.T1} s_{11} \xrightarrow{indus.T1.act1}$$
$$s_{12} \xrightarrow{indus.T2.period} s_{13} \xrightarrow{indus.alloc2.T2} s_{14} \xrightarrow{indus.T2.act1} s_{15} \xrightarrow{indus.T3.period} s_{16} \xrightarrow{indus.alloc2.T3}$$
$$s_{17} \xrightarrow{indus.T1.period} s_{18} \xrightarrow{indus.alloc1.T1} s_{19} \xrightarrow{indus.T3.deadline} s_{20}$$

Pour plus de lisibilité, voici cette séquence sous forme de chronogramme :



En examinant cette séquence d'exécution, on peut constater que le système exécute bien l'allocation $indus.alloc2.T3$ mais aucune action de $T3$ n'est ensuite effectuée, ce qui peut mettre la puce à l'oreille du concepteur et lui permettre de corriger l'erreur dans son modèle.

6.5.2 Analyse quantitative manuelle par raffinages successifs

Ce travail n'a jamais eu pour but de constituer une base de travail pour l'analyse quantitative de systèmes. Néanmoins, il est possible, pour un utilisateur astucieux, de se servir du *model checker* et de son intuition pour trouver des données quantitatives intéressantes.

Par exemple, trouver le temps de réponse maximal d'une tâche est faisable en se servant de la détection de dépassement d'échéance. En effet, le temps de réponse maximal est la durée maximale depuis le réveil d'une tâche jusqu'à sa terminaison. Dans un système ordonnançable, ce temps de réponse est systématiquement inférieur à l'instant échéance de la tâche.

Dans le cas où il ne peut y avoir qu'une instance de tâche par période, une borne maximale du temps de réponse peut être la valeur de la période. Ou, mieux, l'utilisateur peut chercher une borne maximale moins pessimiste en utilisant les méthodes analytiques de calcul du pire temps de réponse. Partir d'une valeur pessimiste de l'échéance pour la restreindre petit à petit revient à raffiner le système, et tester si le raffinage est conservatif. A l'opposé, il est également possible de partir de systèmes ratant leurs échéances (en prenant bien garde à borner le système), et d'élargir la durée maximale de temps de réponse jusqu'à ce que le système ne rate plus cette échéance.

Avoir plusieurs instances de tâches par période est un problème non trivial qui peut être résolu en utilisant une variante de la sémantique différente des *TPN*, la multi-sensibilisation, ou bien en codant ces instances dans le langage Pola lui-même. Dans tous les cas, l'utilisateur *doit* borner le nombre d'instances par période, et par conséquent, il lui est possible de connaître une borne maximale pour les échéances des instances et de procéder comme précédemment.

6.5.3 Comparaison avec l'outil Times

L'outil Times, implémentant une variante du formalisme des automates de tâches, est un outil possédant un langage dédié à la spécification des systèmes de tâches temps réel, comme nous l'avons

abordé dans le portrait esquissé au chapitre 4.

Dans cette section, nous allons tenter de nous positionner plus précisément en étudiant la faisabilité de la spécification des deux cas d'études de ce chapitre à l'aide de cet outil.

Groupes à la Osek Afin de spécifier un tel cas d'étude, nous avons vu qu'il était souhaitable d'ajouter des ressources permettant de dénoter l'exécution courante d'une tâche du groupe.

Le formalisme des automates de tâches ne permet pas de modéliser de ressources, néanmoins, l'outil Times autorise l'utilisation de données dans la description des automates et/ou du comportement en langage C des tâches.

Dans un cas comme dans l'autre, ces nouvelles ressources ou données doivent pouvoir influencer le comportement de la politique d'ordonnancement, puisqu'il est nécessaire de désactiver une partie de la politique selon l'utilisation des ressources (ou données) représentant les groupes. Ceci ne peut pas être fait de manière interne à l'outil, c'est-à-dire par l'utilisation des automates ou d'un code C (à moins de recoder toute la mécanique au sein d'une tâche ... mais cette possibilité nous semble présenter trop d'inconvénients pour nous y attarder), mais peut être accompli de manière externe en étendant l'outil par la prise en charge de cette nouvelle politique.

Partitions à la Arinc 653 Un automate de tâches se charge de réveiller les tâches qui ne sont pas périodiques : à chaque fois qu'une location est atteinte, l'éventuelle tâche qui lui est associée est réveillée. Une fois réveillée, une tâche n'est plus influençable par l'automate : seule la politique sous-jacente permet de suspendre une tâche en cours d'exécution et de la reprendre lorsqu'elle a été suspendue, selon sa priorité.

Il est donc soit possible de coder le contrôleur Arinc 653 en respectant la façon de faire de l'outil, mais alors les tâches du système doivent être codées à la main, ce qui fait perdre tout l'intérêt de l'utilisation d'un tel outil ; soit les tâches des partitions sont écrites à la manière Times (en utilisant éventuellement une donnée pour repérer la partition) et alors là l'outil n'est plus suffisant tel quel, et comme précédemment une nouvelle politique doit être ajoutée qui prend en compte l'interruption d'une tâche appartenant à une partition lorsque cette partition est terminée.

Remarques Dans les cas d'études présentés ici, nous nous sommes limités à des intervalles restreints à une seule valeur à cause de l'indécidabilité latente du formalisme. Times, au lieu de laisser le choix à l'utilisateur d'utiliser un formalisme indécidable, impose de n'avoir que des valeurs précises. Ceci implique que la représentation de l'espace d'état est plus efficace lorsque qu'il y a possibilité de préemption : nous utilisons dans ce cas des polyèdres généraux, alors que Times utilise une structure simplifiée. Il serait bien sûr possible de détecter qu'un modèle n'admet que des intervalles points et qu'ainsi il n'est pas obligatoire d'utiliser une représentation aussi forte que les polyèdres, mais cette problématique se situe surtout au niveau de l'explorateur d'états et sort donc du cadre de notre étude.

Une autre remarque tient au fait que Times utilise une logique plus restreinte que celle que nous utilisons. En effet la logique de Times est un sous-ensemble de CTL[1] ne permettant pas d'imbriquer les modalités. Les formules que nous vérifions pour chaque modèle Pola sont suffisamment simples pour être spécifiées à l'aide de cette logique restreinte :

$$\Box\neg\text{Echéance\_ratée} \equiv A\Box\text{Echéance\_ratée} \quad (6.5)$$

$$\Diamond\text{Fin\_de\_tâche}_{T_i} \equiv A\Diamond\text{Fin\_de\_tâche}_{T_i} \quad (6.6)$$

$$\Box\Diamond\text{Fin\_de\_tâche}_{T_i} \equiv True -\!\!-\!\!> \text{Fin\_de\_tâche}_{T_i} \quad (6.7)$$

Puisque la seule imbrication permise est celle de $-\!\!-\!\!>$, cette restriction de CTL est aussi une restriction de LTL. Toutes les formules exprimables par cette logique sont exprimables en LTL. La formule $\Diamond\Box$ok, utile pour spécifier qu'un système est toujours *ok* au-delà d'un certain état, n'est

[1] les opérateurs utilisables sont : $A\Box$, $A\Diamond$, équivalent à $\Box$ et $\Diamond$ de LTL ; et $E\Box$, $E\Diamond$ équivalent à EG et EF de CTL et finalement $a -\!\!-\!\!> b$ équivalent à $\Box(a \rightarrow \Diamond b)$

doublement pas spécifiable par le langage de requêtes de Times : car elle utilise une imbrication impossible et elle n'est pas du tout spécifiable en *CTL* [CD89].

6.6 Conclusion

Nous avons montré dans ce chapitre que la vérification de systèmes temporisés ne pouvait se faire à partir de certitudes : la dynamique de ces systèmes est généralement tellement complexe que même une personne ayant de l'expérience dans la vérification de ces systèmes pourra être surprise par la complexité et/ou la difficulté de s'assurer du bon fonctionnement de ces systèmes. Il n'existe pas de règles générales, tout le temps applicables, pour optimiser cette vérification, mais des techniques existent pour en réduire la complexité.

Nous avons vu que les grandeurs utilisées avaient de l'importance et qu'il était préférable d'utiliser des nombres multiples entre eux. Il est également préférable de faire en sorte que ces valeurs ne soient pas trop éloignées les unes des autres. De plus, la symétrie des systèmes est une donnée souvent intéressante à prendre en compte. Mais ces paramètres ne sont pas toujours contrôlables, et certains comportements spécifiques peuvent venir casser les généralités précédemment énoncés.

En définitive, l'automatisation de la chaîne de vérification n'est qu'un premier pas, car souvent, pour que l'analyse soit réellement efficace, il convient d'adapter les mécanismes de vérification à une gamme de problèmes encore plus réduite que celle proposée ici. Ce n'est bien sûr possible qu'au prix d'un effort de recherche, qui lui n'est pas automatisable !

Chapitre 7

Conclusion

Si de prime abord la définition des systèmes temps réel parait bien anodine et transporte le faux semblant d'une problématique sans objet, dès que l'on se confronte à la modélisation de tels systèmes, avec seulement quelques tâches, la prise de conscience est rapide : on est vite noyé par les informations temporelles et la dynamique engendrée ne semble plus si banale.

En effet, par une analyse superficielle de la complexité d'un problème temps réel, il peut être facile de se tromper sur sa correction en ne l'analysant que d'un point de vue atemporel : toute propriété valide pour un système atemporel ne l'est pas forcément pour un système temporel. Et les propriétés qui ne sont pas préservées par l'introduction de la dimension temporelle sont potentiellement bien plus compliquées à vérifier puisque des grandeurs denses, manipulables par des opérations compliquées prennent le relais de grandeurs discrètes dont la dynamique peut souvent être réduite à de simples additions et soustractions.

Un parallèle simple, bien que réducteur, peut être fait avec l'arithmétique. Les opérations d'addition, de multiplication, etc. sont très simplement définies. Mais il suffit de les combiner entre elles, d'utiliser des nombres ne s'accordant que peu à un raisonnement mental, pour aboutir à un problème autrement plus complexe. Or, l'utilisation de périodes, par exemple, implique de trouver le plus petit commun multiple, dont le résultat est bien plus difficile à prévoir qu'une addition, pour déterminer la plage temporelle à étudier.

Les systèmes temps réel partagent des facettes identiques : les nombres entrant en jeu dans leur dynamique dominent très souvent leur complexité. De même que l'arithmétique, la composition des opérateurs utilisés, dépendant du modèle de tâche, sont une des sources de la complexité des systèmes temps réels.

Ce que nous avons proposé dans cette thèse peut s'apparenter à une amélioration d'une «calculatrice» pour systèmes temps réel. D'une part, nous avons étendu les concepts de base, par la manipulation d'objets temporels, en proposant le formalisme des réseaux de Petri temporels à permissions/inhibitions, qui est une extension des réseaux de Petri temporels. D'autre part, nous avons proposé un langage, Pola, dont la forme (la syntaxe) est plus proche de la fonction de «calculatrice» que ne pouvaient l'être les *ipTPN*, et pour lequel nous avons donné la sémantique sous forme d'*ipTPN*.

7.1 Contributions

Les réseaux de Petri temporels constituent, avec les automates temporisés, l'une des alternatives les plus adéquates à l'expression des systèmes temps réel. De part leur nature intrinsèquement concurrente, ils permettent de séparer les flots de contrôle et de les combiner très naturellement. L'adjonction d'une information temporelle sur les transitions a introduit une nouvelle dimension *temps*, impliquant la possibilité d'une description plus fine qu'un simple enchaînement atemporel d'évènements.

Néanmoins, si cet ajout a apporté une plus grande marge de manœuvre à la modélisation et a ainsi permis l'expression d'une plus large gamme de systèmes, il introduit de nouveaux problèmes, liés à l'écoulement du temps.

Le modèle des réseaux de Petri étant atemporel, le temps qui s'écoule dans les interstices des séquences d'évènements n'est pas spécifié. Tout se passe au niveau logique : le temps est *logique*. Dès lors que la dimension temps est introduite, il apparaît que l'on peut distinguer le temps logique du temps «physique». Il s'ensuit que l'on peut avoir deux notions de séquence : un évènement peut survenir *avant* un autre tout en intervenant au même *instant temporel*.

Les systèmes temps réel, parce qu'ils doivent réagir à tout changement de l'environnement sont équipés, au niveau physique, de capteurs et d'actionneurs et au niveau logiciel de tâches. Ces systèmes sont donc naturellement découpés en autant de fonctionnalités qu'il y d'actionneurs, de capteurs ou de tâches, ce qui implique une modélisation par composants. Ces composants, en fonction des ressources du système, peuvent être en marche simultanément. L'ordre dans lequel les différents évènements de ces composants doit être traité est important : par exemple, l'ordonnanceur, à un instant donné doit «attendre» d'avoir observé toutes les actions possibles du système avant de pouvoir décider de son intervention, c'est-à-dire qu'à tout instant les actions du système sont prioritaires sur la décision de l'ordonnanceur.

Le formalisme des réseaux de Petri temporels ne permet pas de déterminer la priorité d'un évènement sur un autre de manière suffisamment fine.

Plus généralement, la composition des réseaux de Petri temporels pose problème : la synchronisation des évènements se fait au détriment de la conservation d'une information temporelle pertinente. En effet, l'information temporelle est portée par les transitions : il n'est pas possible de fusionner deux transitions sans perdre l'indépendance temporelle des deux transitions fusionnées et cela sans même considérer les cas pour lesquels cette fusion n'a pas de sens.

Afin d'obtenir une composition conservant le comportement de ses composants, les éléments des composants qui entrent en jeu lors de la composition ne doivent pas porter d'informations temporelles. Respecter cette condition avec les réseaux de Petri temporels est très délicat et pas toujours possible.

C'est pour pallier ces problèmes que nous avons introduit deux nouvelles notions duales contraignant le tir des transitions suivant la valeur des horloges des autres transitions.

Cette extension a pour conséquence d'augmenter strictement le pouvoir expressif des réseaux de Petri temporels et permet de se rapprocher sémantiquement d'un autre formalisme pour la spécification des systèmes temps réel, les automates temporisés. En effet, l'extension proposée intègre dans les réseaux de Petri temporels des bonnes propriétés qui étaient réservés aux automates temporisés, comme la possibilité de composition sans modifier le comportement temporel des composants ; mais il en apporte de moins bonnes, comme la possibilité de blocages temporels. Il est néanmoins important de souligner que l'ajout de priorités dans les automates temporisés, relation équivalente à celle de la relation d'inhibition, oblige dans le cas des automates temporisés à utiliser une structure de données plus complexe et donc plus coûteuse en temps de calcul.

L'autre aspect des travaux de cette thèse concerne l'application de cette nouvelle extension. Afin d'en éprouver les forces et les limites, nous avons conçu Pola, un langage dédié à la spécification de systèmes de tâches temps réel. Ce langage reprend le côté déclaratif de langages comme Aadl, mais fourni également une interface permettant de le relier à des langages plus généralistes afin de prendre en compte des aspects qui ne sont pas gérables par la partie déclarative du langage.

Afin d'être sûr que le langage soit «vérifiable par construction», sa sémantique est donnée en terme de réseaux de Petri temporels étendus notamment par les relations d'inhibition/permission proposées dans cette thèse, mais également par des mécanismes utiles comme les tests arcs et même indispensables comme les arcs chronomètres.

Les limites imposées par le formalisme décrivant la sémantique impliquent donc que le langage soit très ciblé. Nous avons adopté le parti de concevoir un langage fournissant un modèle de tâches couramment reconnu ; supportant l'utilisation de plusieurs ressources, les ressources étant considérées de manière très abstraite par le langage ; permettant la spécification d'un ordonnanceur en précisant la politique selon laquelle une tâche est ordonnancée et en donnant la répartition d'utilisation des ressources par les tâches ; le tout étant encapsulé par une couche système, plusieurs systèmes pouvant être définis au sein d'un même modèle.

La sémantique du langage est ainsi donnée par sa traduction en réseaux de Petri temporels à

contraintes d'inhibition/permission, traduction utilisable pour la vérification par *model checking* : le comportement est extrait de la sémantique du modèle POLA sous une forme analysable par la partie de *model checking* proprement dite. La traduction en *ipTPN* puis la vérification de propriétés génériques telles que l'ordonnançabilité forment une chaîne de vérification dont un prototype a été réalisé au cours de cette thèse.

7.2 Perspectives

L'interfaçage de POLA avec son langage de comportement a été conçu de manière suffisamment lâche afin de permettre à d'autres langages, préférablement de plus haut niveau que les *ipTPN*, de se greffer dans la partie description de comportements spécifiques. Nous pensons notamment à FIACRE, qui est un langage de bien plus haut niveau. La non prise en compte par FIACRE de la suspension/reprise d'écoulement temporel est par contre préjudiciable car elle empêche l'une des approches les plus intéressantes qui serait de redéfinir la sémantique de POLA à l'aide de FIACRE.

Au lieu d'intégrer un langage de haut niveau dans POLA, une autre piste pourrait être d'intégrer POLA directement dans ce langage de haut niveau. Cela peut se situer au niveau de langages comme AADL, mais il risque d'y avoir alors des conflits avec les propriétés (au sens AADL) définissables dans ce langage. Une piste éventuellement moins problématique serait d'étendre FIACRE par les primitives POLA et ainsi d'utiliser FIACRE selon sa fonction première, c'est-à-dire celle de langage intermédiaire (ou pivot).

En plus de son intégration, la forme actuelle du langage est améliorable. En effet, il manque des primitives qu'il nous semble indispensable d'intégrer dans un langage digne du nom de langage dédié aux systèmes de tâches temps réel. Par faute de temps, et parce que bien souvent cela relevait plus d'un problématique de développement que de recherche, nous avons laissé de côté certain côtés qui méritent une attention toute particulière.

L'absence de précédence native, en premier lieu : elle a été justifiée comme présentant suffisamment peu d'intérêt face à la possibilité d'utiliser la partie comportement pour pallier ce manque. L'expérience nous a montré que assez peu d'opérateurs de précédence étaient nécessaires pour arriver à se passer complètement du langage de comportement. Cette problématique est bien sûr à mettre en balance dans l'intégration (ou l'inverse) de POLA dans un langage plus généraliste.

La gestion des valeurs de période face aux valeurs d'échéance peut également poser problème : l'utilisateur peut souhaiter avoir une échéance plus grande que la période de la tâche. Ceci peut soit être correct tel quel, soit l'utilisateur peut vouloir des échéances différentes pour deux instances de tâches différentes en *même temps*. Ceci n'est pas possible actuellement à moins de passer par l'activation d'une autre tâche représentant l'instance s'exécutant en parallèle.

Lorsque le modèle n'est pas ordonnançable, l'accumulation des jetons de réveil est un problème qui n'est pas résolu directement par la partie déclarative. Nous avons vu dans le dernier chapitre, une façon de borner ce comportement. Mais l'utilisateur peut vouloir disposer d'autres méthodes moins radicales qu'un *deadlock* pour repérer des échéances ratées. Selon que l'on étudie des systèmes critiques ou pas, selon le type de temps réel dur/ferme/mou, on peut ne pas vouloir arrêter le système dès l'occurrence de la première erreur.

Lorsque des ressources non préemptables sont partagées entre plusieurs tâches possédant des niveaux de priorités différents, il se passe généralement ce que l'on appelle une inversion de priorités qui peut être la cause de blocages si l'on ne fait pas attention. Pour résoudre ce problème, les protocoles à héritages de priorités ont été conçus, spécifiant des comportements différents face à la prise et/ou à la libération de ressources. Même si certains cas de ces protocoles peuvent être traités par une manipulation des ressources ingénieuse en POLA, le langage se révèle insuffisant (dans sa partie déclarative) pour les modéliser dans le cas général.

Notre objectif principal en terme de vérification de propriétés a été de vérifier la propriété "naturelle" qu'est l'ordonnançabilité d'un système, et de montrer que d'autres propriétés intéressantes pouvaient être vérifiées, comme la vivacité des tâches. Ces propriétés sont automatiquement extraites du modèle sans qu'il y ait besoin d'une quelconque intervention de l'utilisateur. Il serait intéressant

d'aller plus loin, en intégrant au langage la possibilité de donner des spécifications simplifiées des propriétés à vérifier. Cependant, trouver un juste équilibre entre l'approche, celle que nous utilisons actuellement, où l'utilisateur n'a rien à faire, les propriétés «pertinentes» étant choisies pour lui, et celle consistant à donner toute la liberté à l'utilisateur en le laissant écrire lui-même ses formules *LTL*, n'est pas évident et nécessite une étude approfondie de ce qui est spécifique et peut être abstrait de ce qui ne l'est pas au sein d'une propriété.

En ce qui concerne l'analyse des réseaux *ipTPN*, on peut souligner que même si l'extension proposée n'introduit que peu de complexité algorithme supplémentaire, les modèles *ipTPN* possèdent une nette tendance à comporter plus de transitions, car la contrainte sur une transition peut être donnée par plusieurs autres. Les structures de données sont souvent plus volumineuses et le temps de calcul pour la construction d'une classe est en conséquence plus important. Par contre l'utilisation de l'extension permettant d'avoir moins d'artefacts de codage, il n'est pas évident que l'analyse dans sa globalité soit plus longue.

L'utilisation des *ipTPN* implique que dans beaucoup de cas, seules les bornes inférieures des intervalles temporels sont utilisées. Les algorithmes de génération des classes utilisent une technique, dite de *relaxation* pour garantir la terminaison des algorithmes (dans les cas décidables) en relâchant à l'infini le domaine d'une horloge. Les techniques de relaxation ne sont pour l'instant pas implémentées en présence de chronomètres. De plus, il nous semble intéressant d'étudier une telle relaxation en regard des transitions non tirables (boucle $\multimap$) et puisqu'il y a potentiellement plus de transitions dont l'intervalle de tir est non borné (les deux nouvelles relations n'utilisant que la borne inférieure de cet intervalle), peut être serait-il pertinent de ne plus stocker le domaine d'horloge des transitions relaxées en considérant qu'une fois sa borne inférieure dépassée, une horloge dont l'intervalle temporel est non borné n'impose plus de contrainte. Ces intervalles non bornés posent donc à la fois un problème dans le contexte de vérification des arc chronomètres, mais ouvre une perspective intéressante d'optimisation.

A cause de l'indécidabilité de l'accessibilité des états, la vérification d'un système temps réel préemptif n'est pas obligatoirement réalisable de manière automatique. Pour les systèmes qui n'ont pu être vérifiés automatiquement, l'intervention humaine est alors indispensable. Il nous semble intéressant d'étudier les possibilités qu'apportent les techniques de preuves pour diriger la construction des classes, de façon interactive ; ou la méthode duale, utilisant les méthodes de preuves comme base et le *model checking* pour résoudre automatiquement certaines parties décidables de la preuve.

Afin de limiter les problèmes engendrés par l'indécidabilité, il est également possible soit d'utiliser des surapproximations, soit de limiter l'analyse *a priori*, selon un paramètre dont on peut *intuitivement* penser qu'il est discriminant : comme, par exemple, en limitant la taille du graphe ou en spécifiant une borne maximale pour une place, etc., seule l'expérience pouvant guider le choix des paramètres pertinents. Dans tous les cas, l'ensemble des propriétés n'est plus préservé, mais pour celles qui le sont, la procédure reste automatique.

Une formule fausse, que ce cela soit par une analyse exacte ou approximative, doit être retournée de manière intelligible à l'utilisateur. Et si ceci peut paraître évident pour les analyses exactes, c'est encore plus vrai pour les surapproximations, pour lesquelles l'utilisateur peut ne pas savoir pourquoi la formule est invalidée : est-elle intrinsèquement fausse (c'est-à-dire fausse sur le système concret), ou est-ce l'approximation qui introduit la faute ? L'information du contre-exemple peut ainsi être une information cruciale à remonter à un utilisateur pas forcément expert, c'est-à-dire qui ne connaît pas le fonctionnement et les présuppositions faites par les outils d'analyse. C'est donc dans le même langage que celui utilisé pour la modélisation que l'information devrait idéalement remonter.

La sémantique donnée sous forme d'*ipTPN* ne génère que des évènements intelligibles par le concepteur (cf. le contre-exemple donné dans le chapitre précédent), elle respecte donc la problématique introduite ci-dessus. Par contre, certaines places introduites dans la sémantique ne correspondent pas toutes à une donnée intelligible directement par le concepteur. A un autre niveau de problématique, mais tout aussi désagréable, on peut également remarquer qu'un contre-exemple peut être très long et ainsi trop volumineux à étudier. Par exemple, lorsque la méta-période est très longue et que le pas temporel du système est très faible, si une erreur se produit en fin de méta-période, la séquence amenant à la faute sera nécessairement très longue. La séquence contre-exemple est une trace maximale, c'est-

à-dire infinie, qui donne un cycle duquel on n'est obligé de sortir et invalidant la formule. L'extraction d'un cycle qui soit le plus lisible possible n'est pas trivial, et il arrive en effet qu'une séquence contre-exemple soit difficilement exploitable à cause d'une «mauvaise» extraction. Des études sont donc nécessaires pour optimiser la longueur du contre-exemple et d'en améliorer globalement sa pertinence.

Il reste un long chemin à parcourir et c'est tant mieux !

Bibliographie

[ABW95] R. K. Allen, A. Burns, and A. J. Wellings. Sporadic tasks in hard real-time systems. *Ada Lett.*, XV(5) :46–51, 1995.

[ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS 1990)*, pages 414–425. IEEE Computer Society Press, June 1990.

[ACHH93] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata : An algorithmic approach to the specification and verification of hybrid systems. pages 209–229. Springer-Verlag, 1993.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[AF73] Tilak Agerwala and Mike Flynn. Comments on capabilities, limitations and "correctness" of Petri nets. In *ISCA '73 : Proceedings of the 1st annual symposium on Computer architecture*, pages 81–86, New York, NY, USA, 1973. ACM.

[AFH96] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1) :116–146, 1996.

[AFM+02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES - a tool for modelling and implementation of embedded systems. In *Lecture Notes in Computer ScienceTools and Algorithms for the Construction and Analysis of Systems : 8th International Conference, TACAS 2002*, volume 2280, page 460, 2002.

[Age75] Tilak Krishna Mahesh Agerwala. *Towards a theory for the analysis and synthesis of systems exhibiting concurrency.* PhD thesis, The Johns Hopkins University, 1975.

[AM09] Charles André and Frédéric Mallet. Specification and verification of time requirements with CCSL and ESTEREL. *SIGPLAN Not.*, 44(7) :167–176, 2009.

[Aud91] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, 1991.

[BBF+08] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang, and François Vernadat. FIACRE : an intermediate language for model verification in the TOPCASED environment. In *ERTS 2008*, Toulouse France, 2008.

[BCH+05a] B. Bérard, F. Cassez, S. Haddad, D. Lime, and O.H. Roux. When are timed automata weakly timed bisimilar to time Petri nets ? In *25th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2005)*, volume 3821 of *Lecture Notes in Computer Science*, Hyderabad, India, December 2005. Springer.

[BCH+05b] Béatrice Bérard, Franck Cassez, Serge Haddad, Olivier H. Roux, and Didier Lime. Comparison of the expressiveness of timed automata and time Petri nets. In Paul Pettersson and Wang Yi, editors, *Proceedings of the third International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'05)*, volume 3829 of *Lecture Notes in Computer Science*, pages 211–225, Uppsala, Sweden, September 2005. Springer.

[BD91] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3), 1991.

[BDGP98] B. Bérard, V. Diekert, P. Gastin, and A. Petit. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36(2) :145–182, November 1998.

[BDH92] Eike Best, Raymond Devillers, and John G. Hall. The box calculus : a new causal algebra with multi-label communication. *Lecture Notes in Computer Science ; Advances in Petri Nets 1992*, 609 :21–69, 1992.

[Ber07] Antonia Bertolino. Software testing research : Achievements, challenges, dreams. In *FOSE '07 : 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[BF99] Victor A. Braberman and Miguel Felder. Verification of real-time designs : combining scheduling theory with automatic formal verification. In *ESEC/FSE-7 : Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 494–510, London, UK, 1999. Springer-Verlag.

[BFSV03] G. Bucci, A. Fedeli, L. Sassoli, and E. Vicario. Modeling flexible real time systems with preemptive time Petri nets. *Real-Time Systems, Euromicro Conference on*, 0 :279, 2003.

[BH06] Hanifa Boucheneb and Rachid Hadjidj. Using inclusion abstraction to construct Atomic State Class Graphs for time Petri nets. *IJES*, 2(1/2) :128–139, 2006.

[BLRV05] B. Berthomieu, D. Lime, O. H. Roux, and F. Vernadat. Problèmes d'accessibilité et espaces d'états abstraits des réseaux de Petri temporels à chronomètres. In *Modélisation des Systèmes Réactifs 2005, Journal européen des systèmes automatisés*, 2005.

[BM83] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. In R. E. A. Mason, editor, *Information Processing : proceedings of the IFIP congress 1983*, volume 9, pages 41–46. Elsevier Science Publishers, Amsterdam, 1983.

[BPV06] Bernard Berthomieu, Florent Peres, and François Vernadat. Bridging the gap between timed automata and bounded time Petri nets. In *Formal Modeling and Analysis of Timed Systems*, volume Volume 4202/2006 of *Lecture Notes in Computer Science*, pages 82–97. Springer Berlin / Heidelberg, 2006.

[BPV07] Bernard Berthomieu, Florent Peres, and François Vernadat. Model checking bounded prioritized time Petri nets. In *Automated Technology for Verification and Analysis*, 2007.

[BRH06] Patricia Bouyer, Pierre-Alain Reynier, and Serge Haddad. Extended timed automata and time Petri nets. In *ACSD '06 : Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, pages 91–100, Washington, DC, USA, 2006. IEEE Computer Society.

[BRV03] B. Berthomieu, P.-O. Ribet, and F. Vernadat. L'outil TINA – construction d'espaces d'états abstraits pour les réseaux de Petri et réseaux temporels. *Modélisation des Systèmes Réactifs (MSR)*, 2003.

[BV03] B. Berthomieu and F. Vernadat. State class constructions for branching analysis of time Petri nets. In *TACAS 2003*, volume 2619 of *LNCS*, page p. 442. Springer Verlag, 2003.

[BV07] Bernard Berthomieu and François Vernadat. *State Space Abstractions for Time Petri Nets*, chapter Part VI. CRC Press, 2007.

[CCO+04] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. State/event-based software model checking. In *In Integrated Formal Methods*, pages 128–147. Springer-Verlag, 2004.

[CD89] Edmund M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 428–437, London, UK, 1989. Springer-Verlag.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs Workshop, Yorktown Heights, New York, May 1981*, 1981.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[CH92] Sören Christensen and Niels Damgaard Hansen. Coloured Petri nets extended with place capacities, test arcs and inhibitor arcs. In *Applications and Theory of Petri Nets, volume 691 of LNCS*, pages 186–205. Springer Verlag, 1992.

[CL00] Franck Cassez and Kim Larsen. The impressive power of stopwatches. *Lecture Notes in Computer Science*, 1877 :138+, 2000.

[CR06] Franck Cassez and Olivier Roux. Structural translation from time Petri nets to timed automata. *Journal of Systems and Software*, 29(1) :1456–1468, 2006.

[DHLP06] Alexandre David, John Håkansson, Kim G. Larsen, and Paul Pettersson. Model checking timed automata with priorities using DBM subtraction. In *Formal Modeling and Analysis of Timed Systems*, volume Volume 4202/2006 of *Lecture Notes in Computer Science*, pages 128–142. Springer Berlin / Heidelberg, 2006.

[DL07] Alexandre Duret-Lutz. *Contributions à l'approche automate pour la vérification de propriétés de systèmes concurrents*. PhD thesis, Université Pierre etMarie CURIE, 2007.

[FAMdS09] Benoît Ferrero, Charles André, Frédéric Mallet, and Robert de Simone. TIMESQUARE : a software environment for timed systems. In *DATE 2009*, 2009.

[FBDSG07] M. Faugere, T. Bourbeau, R. De Simone, and S. Gerard. MARTE : Also an UML profile for modeling AADL applications. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 359–364, July 2007.

[FGC+06] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. The TOPCASED project : a Toolkit in OPen source for Critical Aeronautic SystEms Design. In *Embedded Real Time Software (ERTS) 2006*, 2006.

[Fin93] Alain Finkel. The minimal coverability graph for Petri nets. In *Papers from the 12th International Conference on Applications and Theory of Petri Nets*, pages 210–243, London, UK, 1993. Springer-Verlag.

[FKPY07] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata : Schedulability, decidability and undecidability. *International Journal of Information and Computation*, 205(8) :1149–1172, August 2007.

[GL06] Volker Gruhn and Ralf Laue. Patterns for timed property specifications. *Electronic Notes in Theoretical Computer Science*, 153(2) :117 – 133, 2006. Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005).

[Had06] Rachid Hadjidj. *Analyse et Validation Formelle des Systèmes Temps Réel*. PhD thesis, Ecole Polytechnique de Montréal, 2006.

[HC68] G. Hughes and M. Cresswell. *An Introduction to Modal Logic*. Methuen, 1968.

[HKPV95] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata ? In *STOC '95 : Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 373–382, New York, NY, USA, 1995. ACM.

[HMP92] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In *Proceedings of the Real-Time : Theory in Practice, REX Workshop*, pages 226–251, London, UK, 1992. Springer-Verlag.

[JLL77] Neil D. Jones, Lawrence H. Landweber, and Y. Edmund Lien. Complexity of some problems in Petri nets. *Theor. Comput. Sci.*, 4(3) :277–299, 1977.

[KC05] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *ICSE '05 : Proceedings of the 27th international conference on Software engineering*, pages 372–381, New York, NY, USA, 2005. ACM.

[KM69] R. M. Karp and R. E. Miller. Parallel program schemata. *Journ. Computer and System Sciences 3*, (2) :147–195, May, 1969.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4) :255–299, 1990.

[Lan66] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3) :157–166, 1966.

[LHHC05] Shang-Wei Lin, Pao-Ann Hsiung, Chun-Hsian Huang, and Yean-Ru Chen. Model checking prioritized timed automata. In *Automated Technology for Verification and Analysis*, 2005.

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

[MA08] Frédéric Mallet and Charles André. UML/MARTE CCSL, SIGNAL and Petri nets. Research Report RR-6545, INRIA, 2008.

[MAJ09] Frédéric Mallet, Charles André, and Deantoni Julien. Executing AADL models with UML/MARTE. In *Int. Conf. Engineering of Complex Computer Systems - ICECCS'09*, pages pp. 371–376, Potsdam Allemagne, 06 2009.

[Mer74] P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, University of California, 1974.

[Mig99] Jörn Migge. *L'ordonnancement sous contraintes temps réel : un modèle à base de trajectoires.* PhD thesis, Université de Nice Sophia Antipolis, 1999.

[Min61] Marvin L. Minsky. Recursive unsolvability of post's problem of "tag" and other topics in theory of Turing machines. *The Annals of Mathematics*, 74(3) :437–455, 1961.

[MNP06] Oded Maler, Dejan Nickovic, and Amir Pnueli. From MITL to timed automata. In FORMATS, volume 4202 of *LNCS*, pages 274–289. Springer, 2006.

[NOSY93] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine1. An approach to the description and analysis of hybrid systems. In Springer, editor, *Hybrid Systems*, volume LNCS 736, pages 149–178, 1993.

[OMG08] Inc Object Management Group. A UML profile for MARTE : Modeling and analysis of real-time embedded systems,beta 2, 2008.

[OMG09a] Inc Object Management Group. Unified modelling language (UML) 2.2 infrastructure, 02 2009.

[OMG09b] Inc Object Management Group. Unified modelling language (UML) 2.2 superstructure, 02 2009.

[OSE] OSEK/VDX. OSEK/VDX – operating system. version 2.2.3.

[PBV09] Florent Peres, Bernard Berthomieu, and François Vernadat. Composer des réseaux de Petri temporels. In *Journal européen des systèmes automatisés — Modélisation des systèmes réactifs, MSR 2009*, volume 43. Lavoisier, 2009.

[Pet62] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[Pet66] C. A. Petri. Communication with automata. Technical Report RADC-TR-65-377, Vol. 1, Suppl. 1, Griffiss Air Force Base, 1966.

[PHD01] Julio L. Medina Pasaje, Medina González Harbour, and Jose M. Drake. MAST real-time view : A graphic UML tool for modeling object-oriented real-time systems. *Real-Time Systems Symposium, IEEE International*, 0 :245, 2001.

[Pom02] Franck Pommereau. *Modèles composables et concurrents pour le temps-réel*. PhD thesis, Université Paris 12, 2002.

[Pri67] Arthur Prior. *Past, Present and Future*. Oxford, 1967.

[Ray03] Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.

[RD02] Olivier H. Roux and Anne-Marie Déplanche. A t-time Petri net extension for real-time task scheduling modeling. *Journal Européen Des Systèmes Automatisés*, 2002.

[RL04] Olivier H. Roux and Didier Lime. Time Petri nets with inhibitor hyperarcs. formal semantics and state space computation. In *25th international conference on theory and application of Petri nets (ICATPN 2004)*, 2004.

[Rok93] T. G. Rokicki. *Representing and Modeling Circuits*. PhD thesis, Stanford University, 1993.

[RSL88] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 259–269, Dec 1988.

[SAE04] SAE standard. Architecture analysis & design language (aadl), 2004. Numéro de standard AS5506A, deuxième version en 2009.

[San04] Davide Sangiorgi. Bisimulation : From the origins to today. *Logic in Computer Science, Symposium on*, 0 :298–302, 2004.

[SLNM05] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with AADL. *Ada Lett.*, XXV(4) :1–10, 2005.

[Spi01] Diomidis Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1) :91–99, February 2001.

[Sta88] John A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10) :10–19, October 1988. U. Mass.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages : an annotated bibliography. *SIGPLAN Not.*, 35(6) :26–36, 2000.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem—overview of the methods and survey of tools. 7(3), 2008. ACM Transactions on Embedded Computing Systems (TECS).

[Wie06] Freek Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006.

[Yi90] Wang Yi. Real-time behaviour of asynchronous agents. In *CONCUR '90 : Proceedings on Theories of concurrency : unification and extension*, pages 502–520, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[YR98] Tomohiro Yoneda and Hikaru Ryuba. CTL model checking of time Petri nets using geometric regions. In *IEICE Trans.*, volume Vol.E81-D, pages pp.297–396, 1998.

Réseaux de Petri temporels à inhibitions/permissions – Application à la modélisation et vérification de systèmes de tâches temps réel

Résumé Les systèmes temps réel (STR) sont au cœur de machines souvent jugés critiques pour la sécurité : ils en contrôlent l'exécution afin que celles-ci se comportent de manière sûre dans le contexte d'un environnement dont l'évolution peut être imprévisible. Un STR n'a d'autre alternative que de s'adapter à son environnement : sa correction dépend des temps de réponses aux stimuli de ce dernier.

Il est couramment admis que le formalisme des réseaux de Petri temporels (RdPT) est adapté à la description des STR. Cependant, la modélisation de systèmes simples, ne possédant que quelques tâches périodiques ordonnancées de façon basique se révèle être un exercice souvent complexe.

En premier lieu, la modélisation efficace d'une gamme étendue de politiques d'ordonnancements se heurte à l'incapacité des RdPT à imposer un ordre d'apparition à des évènements concurrents survenant au même instant. D'autre part, les STR ont une nette tendance à être constitués de caractéristiques récurrentes, autorisant une modélisation par composants. Or les RdPT ne sont guère adaptés à une utilisation compositionnelle un tant soit peu générale. Afin de résoudre ces deux problèmes, nous proposons dans cette thèse Cifre – en partenariat entre Airbus et le Laas-Cnrs – d'étendre les RdPT à l'aide de deux nouvelles relations, les relations d'inhibition et de permission, permettant de spécifier de manière plus fine les contraintes de temps.

Afin de cerner un périmètre clair d'adéquation de cette nouvelle extension à la modélisation des systèmes temps réel, nous avons défini Pola, un langage spécifique poursuivant deux objectifs : déterminer un sous-ensemble des systèmes temps réel modélisables par les réseaux de Petri temporels à inhibitions/permissions et fournir un langage simple à la communauté temps réel dont la vérification, idéalement automatique, est assurée par construction. Sa sémantique est donnée par traduction en réseaux de Petri temporels à inhibitions/permissions. L'explorateur d'espace d'états de la boite à outils Tina a été étendu afin de permettre la vérification des descriptions Pola.

Mots clés Systèmes temps réel, *model checking*, réseau de Petri temporel, langage spécifique à un domaine

Forbid/Allow time Petri nets – Application to the modeling and checking of real time tasks systems

Abstract Real time systems (RTS) are at the core of safety critical devices : they control the devices' behavior in such a way that they remain safe with regard to an unpredictable environment. A RTS has no other choices than to adapt to its environment : its correctness depends upon its response time to the stimuli stemming from the environment.

It is widely accepted that the Time Petri nets (TPN) formalism is adapted to the description of RTS. However, the modeling of simple systems with only a few periodic tasks scheduled according to a basic policy remains a challenge in the worst case and can be very tedious in the most favorable one.

First, we put forward some limitations of TPN regarding the modeling of a wide variety of scheduling policies, coming from the fact that this formalism is not always capable to impose a given order on events whenever they happen at the same time. Moreover, RTS are usually constituted of the same recurring features, implying a compositional modeling, but TPN are not well adapted to such a compositional use. To solve those problems we propose in this Cifre thesis – in partnership with Airbus and the Laas-Cnrs – to extend the formalism with two new dual relations, the *forbid* and *allow* relations so that time constraints can be finely tuned.

Then, to assess this new extension for modeling of real time systems, we define Pola, a specific language aimed at two goals : to determine a subset of RTS which can be modeled with forbid/allow time Petri nets and to provide a simple language to the real time community which, ideally, can be checked automatically. Its semantics is given by translation into forbid/allow Time Petri nets. The state space exploration tool of the Tina toolbox have been extended so that it can model check Pola descriptions.

Keywords Real time systems, model checking, time Petri nets, domain specific language