Vérification formelle de systèmes. Contribution à la réduction de l'explosion combinatoire

Pierre-Olivier Ribet

THESE

Préparée au

Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS

En vue de l'obtention du titre de
Docteur de l'Institut National des Sciences Appliquées de Toulouse

Par

Pierre-Olivier RIBET
Ingénieur ENSEIRB

Vérification formelle de systèmes
Contribution à la réduction de l'explosion combinatoire

Soutenue le 29 Juin 2005 devant le jury :

M. Serge HADDAD	Rapporteur
M. Charles ANDRÉ	Rapporteur
M. Michel DIAZ	Directeur de thèse
M. François VERNADAT	Examinateur
M. Pierre MICHEL	Examinateur
M. Jean-Marie FARINES	Examinateur
M. Patrick FARAIL	Examinateur

Rapport LAAS

Remerciements

Ce travail de thèse n'aurait pas pu voir le jour sans le soutient et l'aide de mon directeur de thèse François Vernadat, aussi je tiens à le remercier en premier lieu pour ce rôle central qu'il a joué, pour sa disponibilité et tout le serieux et le coeur qu'il a mis dans l'encadrement de mon travail.

Je remercie Bernard Berthomieu qui m'a grandement aidé dans l'implémentation et le tests des techniques développées dans le cadre de ma thèse.

Je tiens également à remercier particulièrement Jean-Marie Farines pour avoir été ami, voisin de bureau, collègue de recherche et membre du jury de ma thèse. Également pour nous avoir envoyé de nombreux stagiaires brésiliens avec qui j'ai partagé des moments très agréables ; Juan, Adriano et Tatiana se reconnaîtront ici.

Enfin je remercie toute ma famille qui m'a soutenu tout le long de ce travail thèse, mes parents qui m'ont toujours fait confiance et je dédie ce manuscrit à Bruno qui n'a jamais voulu croire que je le finirais !

Contents

Introduction

A l'ère de l'information, il est tout à fait logique que l'informatique se généralise. Là où électronique et mécanique dédiées permettaient de traiter une information, l'informatique permet de complexifier et surtout de rendre modulable le traitement. De nombreux systèmes ont donc été « informatisés » pour augmenter les fonctionnalités offertes par ceux-ci, les rendre plus modulables, tout en diminuant les coûts de développement et de mise à jour. Même les systèmes dit « critiques » utilisent de plus en plus l'informatique (domaine automobile, avionique...).

Une erreur de programmation, plus communément appelée « bug informatique » est sans conséquence pour de nombreux systèmes ou tout au plus entraîne l'agacement de l'utilisateur. Par contre pour les systèmes critiques, assurer un « bon » fonctionnement est primordial. La sécurité des personnes ou du matériel doit être garantie, indépendamment des fonctionnalités offertes. Il est donc indispensable de pouvoir assurer le fonctionnement correct de certains systèmes.

Les tests de systèmes sont utiles et indispensables puisqu'ils permettent d'augmenter la confiance que l'on a en ceux-ci. Cependant il est très difficile, voire impossible, que les tests couvrent la totalité du code d'un système complexe. Pourtant tant qu'il reste une partie de code non testée, on ne peut pas garantir le fonctionnement.

L'objectif d'une vérification est d'analyser le système pour garantir son fonctionnement. Pour ce faire, une description du système est nécessaire, ainsi qu'une description d'un « bon » fonctionnement. Une description du système en langage naturel ne sera souvent pas précise et de nombreuses interprétations seront possibles. Par conséquent la vérification d'un tel système serait faite par rapport à une interprétation donnée. Pour résoudre ce problème on utilise un langage formalisé (appelé langage formel), avec une interprétation (appelée sémantique) définie et précise. Pour décrire les comportements corrects du système, plusieurs solutions sont possibles. Une solution consiste à écrire les propriétés que le système doit satisfaire. Comme pour la description du système, les propriétés exprimées ne doivent avoir qu'une seule interprétation possible. Elles seront donc également écrites dans un langage formel muni lui aussi d'une sémantique précise.

La vérification formelle d'un système se décompose en plusieurs étapes. Il faut d'une part une phase de description dans un langage formel (i.e. langage non ambigu) du système et d'autre part une « traduction » du cahier des charges en « spécifications » que doit respecter le système. La phase de vérification consiste alors à vérifier que le comportement du système décrit satisfait les propriétés spécifiées. Le principe de la vérification formelle est donc relativement simple à utiliser. Les parties « critiques » (pour garantir le fonctionnement du système) sont localisées dans la phase de traduction du cahier des charges en spécifications.

Malheureusement la mise en oeuvre pratique n'est pas aussi simple ; en effet le graphe représentant les comportements possibles du système augmente de façon exponentielle par rapport à la taille du système. Ce qui rend toute vérification d'un système de taille conséquente très difficile, voire impossible. Les progrès technologiques font progresser très rapidement la puissance de calcul des ordinateurs et supercalculateurs, mais même si la puissance est doublée ou multipliée par dix tous les ans, les progrès technologiques ne permettent pas à eux seuls d'aborder la vérification

de systèmes complexes, car la complexité de la vérification augmente de façon exponentielle par rapport à la taille du système et les systèmes à vérifier sont de plus en plus complexes.

Puisque le problème ne peut pas être résolu par un progrès technique, la recherche doit trouver une réponse par une solution théorique. Les solutions théoriques permettant de maîtriser cette explosion peuvent être cataloguées en deux parties : la gestion de l'explosion ou des méthodes de réduction du graphe. Le travail présenté dans cette thèse s'intéresse plus particulièrement à cette seconde catégorie. Pour réussir à réduire efficacement la taille du graphe construit il faut s'intéresser aux causes de son explosion qui peuvent être diverses.

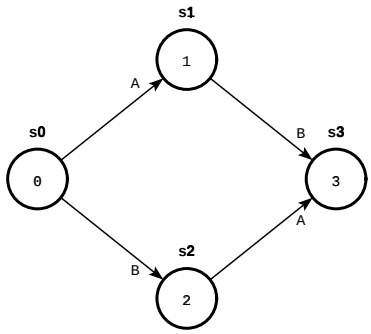


Figure 1: Entrelacement d'action : losange

Nous nous sommes d'abord intéressés à l'explosion due à la représentation du parallélisme par entrelacement d'actions : pour deux actions parallèles A et B, on considère d'une part l'exécution $A.B$ et d'autre part l'exécution $B.A$. Pour des transitions confluentes les deux séquences d'exécution conduisent au même état, on parle alors de « losange » représenté figure 1. Par conséquent dans le cas de plusieurs actions en parallèle, la combinatoire peut être très importante : on parle alors d'explosion combinatoire du nombre d'états. Le graphe construit représente un hypercube dont la dimension est égale au nombre d'actions en parallèle. Des techniques basées sur les ordres partiels permettent de limiter le nombre d'exécutions considérées. On s'intéresse en particulier à deux types d'approches :

- techniques par choix : ces techniques ne choisissent et n'explorent qu'un sous-ensemble des entrelacements possibles. Nous présentons les ensembles persistants, proviso et amples dans la thèse.
- techniques par pas : ces techniques construisent des « pas » qui sont des ensembles de transitions que l'on peut tirer « simultanément » afin d'éviter toutes les combinaisons d'entrelacement possibles des transitions du pas. Nous présentons la technique « Simultaneous Reachability Analysis » (SRA), l'approche définie par Magniette et les graphes de pas couvrants.

Ces différentes techniques de réductions du graphe construit doivent être mises en relation avec la classe de propriétés préservées. En effet suivant le type de propriété que l'on souhaite vérifier, il faut conserver plus ou moins d'informations. On peut faire le parallèle avec la compression de données non conservative. En effet, certains algorithmes sont paramétrés par la quantité d'information que l'on souhaite conserver. Par exemple dans le cas de la compression d'images, on peut compresser plus ou moins en fonction de la « qualité » (i.e. quantité d'information) souhaitée. Dans la version standard des graphes de pas couvrants, les propriétés préservées sont les blocages et les traces maximales (modulo des permutations que nous expliquerons). Nous avons donc travaillé à construire une variante de la technique des graphes de pas couvrants qui permet de conserver une autre classe de propriétés : les formules de la logique temporelle linéaire.

Nous montrons qu'en fait, nous préservons un ensemble plus large constitué par l'ensemble des propriétés bégayement invariantes.

L'observation des réductions obtenues par les différentes techniques dites par « choix » et par « pas » citées précédemment, montre que la réduction n'est pas intrinsèque à la méthode utilisée, mais dépend aussi grandement du système étudié. On peut faire à nouveau le parallèle avec la compression de données ; de nombreux algorithmes existent pour compresser les données et chacun d'eux est plus ou moins efficace en fonction de la structure des données. La structure des données est souvent liée au type des données : fichier son, fichier texte, fichier exécutable, fichier image, fichier video...

Pourtant dans le cas des systèmes que nous étudions, le classement n'est pas toujours aussi évident. Certaines caractéristiques (comme le degré de parallélisme) nous permettent de cataloguer les systèmes, mais il est de façon générale difficile de savoir a priori quelle méthode va être la plus efficace.

Fort de ces constats, notre approche a été d'essayer de faire collaborer deux techniques de réduction. Nous avons choisi une technique de la première catégorie (les ensembles persistants) et une technique de la seconde (les graphes de pas couvrants). L'objectif recherché est double ; d'une part on cherche à cumuler les bénéfices des deux techniques et d'autre part à avoir une technique qui obtient des réductions efficaces sur un plus grand nombre de systèmes.

Une autre cause d'explosion à laquelle nous nous sommes intéressés, concerne les systèmes avec contraintes temporelles. L'étude du comportement d'un système temporisé est difficile car il faut distinguer des comportements identiques qui se produisent à des moments différents. Pour essayer de maîtriser l'explosion du graphe liée à la présence de contraintes temporelles, nous proposons une technique de sur-approximation du comportement du système. Considérer une sur-approximation ou abstraction du système permet de construire un comportement qui peut être représenté de façon plus concise. Ces approches nous permettent de vérifier des propriétés de LTL. Lorsque l'étude de la sur-approximation ne permet pas de conclure, nous avons défini une approche permettant d'utiliser les résultats obtenus par la sur-approximation pour analyser le système.

Guide de lecture

Dans un premier chapitre nous donnons les trois briques essentielles à la vérification : le modèle, la propriété et le processus de vérification. Pour chacune de ces briques nous avons choisi un représentant. Concernant le modèle opérationnel nous avons considéré les réseaux de Petri et les réseaux de Petri temporels. On décrit alors comment le graphe des comportements du système peut être construit. Pour les propriétés nous nous sommes principalement intéressés à des propriétés générales et des propriétés de la logique LTL dont la sémantique est rappelée. Enfin nous décrivons la procédure de vérification d'une propriété LTL sur un système.

Le second chapitre présente le problème de l'explosion combinatoire et présente un tour d'horizon des différentes techniques permettant de le combattre. Nous nous focaliserons plus particulièrement sur l'une d'entre elles : « le graphe des pas couvrants » dans le chapitre suivant.

Dans le quatrième chapitre nous décrivons une technique basée sur les pas, permettant la vérification de formules bégayement invariantes.

Le cinquième chapitre est focalisé sur la propriété de blocage. Pour construire un graphe réduit, nous avons combiné deux techniques : la technique des ensembles persistants et la technique des

graphes de pas couvrants.

Dans le dernier chapitre nous étudions la vérification de systèmes temporels. Nous définissons une sur-approximation et montrons comment on peut mener une vérification sur celle-ci pour diminuer l'explosion du nombre d'états due aux contraintes temporelles.

Chapter I

Vérification de modèles

1 Modèles de comportement

1.1 Réseau de Petri

1.1.1 Définition d'un réseau de Petri, représentation textuelle et graphique

Définition 1 *Réseau de Petri*

Un réseau de Petri places-transitions est un quadruplet $R = \langle P, T, Pre, Post \rangle$*, dans lequel* $P = \{p_1, p_2 \ldots, p_n\}$ *est un ensemble de places,* $T = \{t_1, t_2 \ldots, t_m\}$ *un ensemble de transitions,* $Pre : P \times T \mapsto \mathbb{N}$ *une fonction poids d'incidence avant,* $Post : P \times T \mapsto \mathbb{N}$ *une fonction poids d'incidence arrière.*

Pour chaque transition $t \in T$, on définit les abréviations suivantes :
$^\bullet t = \{p \in P : Pre(p,t) \neq 0\}$
$t^\bullet = \{p \in P : Post(p,t) \neq 0\}$
$^\bullet t^\bullet = {}^\bullet t \cup t^\bullet$

Définition 2 *Réseau de Petri marqué*

Un réseau de Petri places-transitions marqué se définit par un couple (R, m) *dans lequel* R *est un réseau de Petri places-transitions et* $m : P \mapsto \mathbb{N}$ *une application appelée marquage, qui spécifie le nombre de jetons présents dans chaque place.*

Une représentation graphique peut être associée à un réseau de Petri places/transitions marqué. Les places sont représentées par des cercles. La valeur de $m(p)$ est représentée par des « jetons » à l'intérieur de la place. Enfin les transitions sont représentées par un rectangle et des arcs qui relient ce rectangle aux places de $^\bullet t^\bullet$. Le sens de ces arcs permet de faire la différence entre $^\bullet t$ et $t^\bullet$. Si $Pre(p,t) \neq 0$ alors il y a un arc de la place p à la transition t. Si $Post(p,t) \neq 0$ alors il y a un arc de la transition t à la place p. A chaque arc est associé un poids qui correspond à la valeur de $Pre(p,t)$ ou $Post(p,t)$. Pour la représentation graphique, le poids est inscrit à côté de l'arc associé. Si ce poids est égal à 1, sa valeur est implicite et n'est pas indiquée. Un exemple de représentation graphique et textuelle est donné dans la figure I.1.

1.1.2 Sémantique d'un réseau de Petri

Un réseau de Petri est caractérisé d'une part par sa structure statique constituée par ses places et ses transitions décrivant les règles d'évolution du système. Et d'autre part par l'état

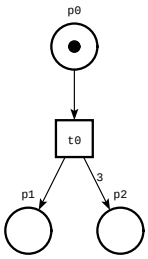
Définition	Représentation graphique
$P = \{p_0, p_1, p_2\}$ $T = \{t_0\}$ $Pre(p_0, t_0) = 1$ $Post(p_1, t_0) = 1$ $Post(p_2, t_0) = 3$ $m(p_0) = 1$	

Figure I.1: Exemple d'un réseau de Petri

du système à un instant donné, défini par le marquage, qui évolue lors du tir de transitions. Les transitions ne peuvent être tirées que si elles sont sensibilisées.

Définition 3 *Transition sensibilisée*
Une transition t d'un réseau de Petri R est sensibilisée dans un marquage m, ssi

$$\forall p \in P, m(p) \geq Pre(p, t)$$

On note alors $m \overset{t}{\twoheadrightarrow}$.

Cette condition de « sensibilisation » signifie que toutes les places de ${}^\bullet t$ représentant les pré-conditions de la transition t ont suffisamment de jetons (i.e. au moins autant que la valeur de $Pre(p, t)$). Dans l'exemple de la figure I.1, la transition t_0 est sensibilisée dans le marquage initial.

Définition 4 *Ensemble des transitions sensibilisées*
L'ensemble E ou $Enabled$ des transitions sensibilisées dans un marquage m est défini par :

$$Enabled(m) = \{t \in T | \forall p \in P, m(p) \geq Pre(p, t)\}$$

1.1.3 Sémantique de l'entrelacement

Les systèmes concurrents ont la spécificité de pouvoir avoir plusieurs transitions sensibilisées simultanément. Une sémantique qui est classiquement utilisée est la **sémantique de l'entrelacement** : une seule transition est tirable en même temps. Cette solution présente l'avantage de considérer tous les cas possibles : si un marquage sensibilise n transitions, alors il aura n marquages successeurs (éventuellement identiques). Cependant, comme nous le verrons plus loin, cette sémantique conduit à une explosion combinatoire lorsque le système comporte de nombreux composants concurrents.

D'autres sémantiques sont également possibles, mais nous ne les envisagerons pas dans ce document. On peut cependant citer la sémantique dite du « vrai parallélisme » qui présente l'intérêt de ne pas multiplier le nombre de configurations lorsque de nombreux composants s'exécutent en parallèle. Malheureusement à notre connaissance, il n'existe pas de formalisme d'expression et de vérification de propriétés pour valider un système décrit dans un formalisme de « vrai parallélisme ».

La sémantique utilisée étant fixée, il est nécessaire de définir le marquage obtenu après le tir d'une transition.

Définition 5 *Tir d'une transition*

Dans un marquage m le tir d'une transition sensibilisée t conduit à un nouveau marquage m' caractérisé par :

$$\forall p, m'(p) = m(p) - Pre(p,t) + Post(p,t)$$

On note alors : $m \overset{t}{\rightarrow\!\!\!\!\!\rightarrow} m'$.

Le marquage m' est obtenu à partir du marquage m dans lequel les jetons se trouvant dans les places ${}^\bullet t$ ont été « consommés » par le tir de la transition t et des jetons ont été « produits » dans les places $t^\bullet$. Le nombre de jetons produits ou consommés dans chaque place est donné par le poids de l'arc, i.e. la valeur de $Pre(p,t)$ ou $Post(p,t)$ pour une place p.

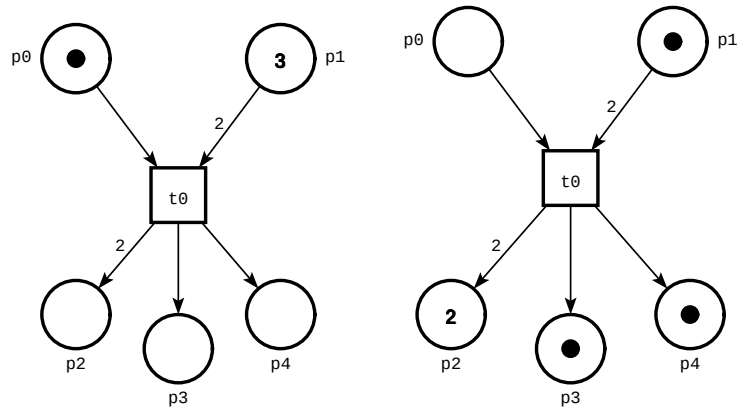


Figure I.2: Exemple du tir d'une transition. Avant et après le tir

La figure I.2 montre un exemple de tir d'une transition. Pour plus de lisibilité, lorsqu'une place comporte plus d'un jeton, les jetons ne sont pas dessinés, mais le nombre de jetons est inscrit au centre de la place. On note sur cet exemple que le nombre total de jetons dans le système peut varier ; le tir d'une transition peut augmenter ou diminuer le nombre global de jetons. Dans cet exemple le marquage initial contenait 4 jetons. Après le tir de la transition t_0, le marquage du réseau de Petri contient 5 jetons.

On étend la définition du tir d'une transition aux séquences de transitions.

Définition 6 *Tir d'une séquence de transitions*

Une séquence de transitions $w = t_1 \dots t_n$ est tirable depuis le marquage m et conduit au marquage m' s'il existe une séquence de marquages $m_1.m_2 \dots m_n$ telle que :

$$m \overset{t_1}{\rightarrow\!\!\!\!\!\rightarrow} m_1 \overset{t_2}{\rightarrow\!\!\!\!\!\rightarrow} \dots \overset{t_n}{\rightarrow\!\!\!\!\!\rightarrow} m_n = m'$$

On écrira alors $m \overset{w}{\Rightarrow} m'$.

On définit le support d'une séquence de transitions comme l'ensemble des transitions apparaissant dans la séquence.

Définition 7 *Support d'une séquence de transitions*

Soit $||\ \ ||$, l'application de $T^* \mapsto \mathcal{P}(T)$ définie par :
$$\begin{array}{ll} ||\epsilon|| & = \emptyset \\ ||t.w|| & = \{t\} \cup ||w|| \end{array}$$

1.2 Graphe d'accessibilité

1.2.1 Structure de Kripke étiquetée et système de transitions

Le comportement d'un réseau de Petri va être représenté à l'aide d'une structure de graphe (Système de Transitions Etiquetées). Cette structure va permettre d'analyser le système par exemple avec des propriétés décrites par une formule LTL. Comme nous le verrons dans la section suivante, une formule LTL est définie sur une séquence d'ensembles de propositions d'états, nous avons donc aussi besoin d'un étiquetage sur les états.

Nous allons utiliser une Structure de Kripke Étiquetée (SKE) qui est une structure de Kripke classique [Kri71] pour laquelle la relation de transition est étiquetée. Cette structure englobe les deux suivantes : à partir d'un SKE, on peut définir un Système de Transitions Étiquetées (STE) en supprimant les propositions sur les états ou une Structure de Kripke (SK) en supprimant les étiquettes sur les transitions.

Définition 8 *Structure de Kripke Étiquetée (SKE)*

$\mathcal{SKE} = \langle AP, T, S, s_0, \{\overset{t}{\longrightarrow}\}_{t\in T}, \nu\rangle$ *est définie par :*

- AP *est un ensemble de propositions atomiques*
- T *est un alphabet fini d'événements*
- S *est un ensemble d'états*
- s_0 *un état particulier de* S *(état initial)*
- $\overset{t}{\longrightarrow}$ *est une relation incluse dans* $S \times T \times S$
- $\nu : S \to 2^{AP}$ *est un étiquetage qui associe à chaque état* s *de* S*, l'ensemble* $\nu(s)$ *des propositions atomiques vérifiées en* s*.*

Définition 9 *Système de Transitions Étiquetées et Structure de Kripke*

Soit $\mathcal{SKE} = \langle AP, T, S, s_0, \{\overset{t}{\longrightarrow}\}_{t\in T}, \nu\rangle$ *une structure de Kripke étiquetée alors :*

- le quadruplet $\mathcal{STE} = \langle S, s_0, T, \{\overset{t}{\longrightarrow}\}_{t\in T}\rangle$ *est un* **Système de transitions étiquetées** *(STE).*

- le quintuplet $\mathcal{SK} = \langle AP, S, s_0, \to, \nu\rangle$ *est une* **Structure de Kripke** *(SK).*

1.2.2 Graphe d'accessibilité d'un réseau de Petri marqué

Définition 10 *Ensemble des marquages accessibles*

Soit un réseau de Petri places/transitions (R, m_0) *avec* m_0 *comme marquage initial. L'ensemble des marquages accessibles ou ensemble d'accessibilité, noté* $S(R, m_0)$ *ou* S*, se définit comme le plus petit ensemble tel que :*

$$m_0 \in S \text{ et } \forall m \in S,\ \text{Si } m \overset{t}{\twoheadrightarrow} m' \text{ Alors } m' \in S$$

Définition 11 *Graphe d'accessibilité*

Soit $R = \langle P, T, Pre, Post\rangle$ *un réseau de Petri et* m_0 *un marquage initial. Le graphe des marquages accessibles (ou graphe d'accessibilité) de* (R, m_0) *est défini comme le système de transitions étiquetées* $STE(R, m_0) = \langle S, s_0, T, \{\overset{t}{\longrightarrow}\}_{t\in T}\rangle$ *dont les états sont les marquages accessibles* $(S = S(R, m_0))$*, l'état initial* $s_0 = m_0$ *et dont les arcs, étiquetés par les noms des transitions, sont définis par la relation de tir entre les marquages.*

$$\forall m, m' \in S,\ (m, t, m') \in \{\overset{t}{\longrightarrow}\}_{t\in T} \text{ si et seulement si } m \overset{t}{\twoheadrightarrow} m'$$

Pour définir une structure de Kripke étiquetée, on ajoute la définition de la fonction de valuation $\nu(s)$ d'un état par l'ensemble des places marquées dans l'état $s : \nu(m) = \{p \in P | m(p) \geq 0\}$. Il est également possible d'étendre les propriétés au multi-marquage, en remplaçant l'ensemble de la valuation par un multi-ensemble pour représenter le nombre de jetons dans chaque place. Nous donnons un exemple de propriétés permises par cette extension dans la partie concernant le model checking.

Définition 12 *Réseau de Petri non Borné*
Un réseau de Petri (R, m_0) est **non borné** *ssi :*

$$\forall n \in I\!N, \exists m \in S(R, m_0), \exists p \in P \text{ tel que } m(p) > n$$

Un réseau de Petri est non borné lorsque au moins une de ses places peut avoir un nombre de jetons non borné. Le réseau de Petri de la figure I.3 est un exemple classique d'écrivain/lecteur. L'écrivain peut écrire autant de messages qu'il le souhaite, ces messages sont transmis au lecteur qui les consomme quand il le souhaite également. Ce réseau est donc non borné.

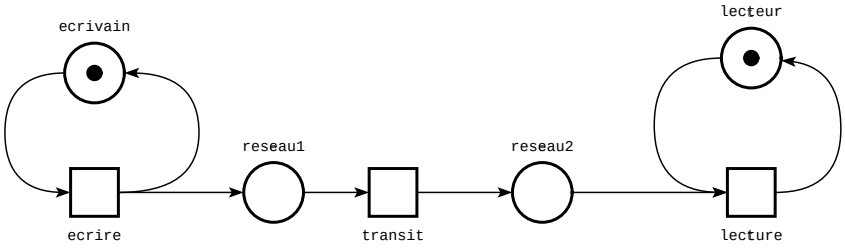


Figure I.3: Écrivain/Lecteur non borné

Un réseau de Petri non borné a donc un nombre infini de marquages accessibles et un graphe d'accessibilité infini également.

Propriété 1 *Finitude du graphe d'accessibilité*
Le graphe d'accessibilité $STE(R, m_0)$ est fini ssi le réseau de Petri (R, m_0) est borné.

Dans la pratique une place non bornée signifie souvent un problème de débordement de variable, de file... l'exemple précédent peut être borné par exemple par une gestion de flux, qui permet d'obtenir le réseau structurellement borné de la figure I.4 : la somme des places $reseau1 + reseau2 + capacite$ est constante.

Le caractère borné dans un réseau de Petri est décidable. L'algorithme de Karp et Miller[KM69] permet non seulement de savoir si le réseau est borné, mais de déterminer les places non bornées. La procédure de construction du graphe d'accessibilité termine uniquement si le graphe d'accessibilité est fini. Pour cette raison le terme de **procédure** est plus approprié que le terme **d'algorithme**. Par abus de langage nous utilisons le terme « algorithme » dans la suite, puisque cette procédure n'est utilisée que pour les réseaux de Petri bornés.

1.2.3 Construction du graphe d'accessibilité

La construction du graphe d'accessibilité est un préambule indispensable à de nombreuses méthodes de vérification. Il permet en effet de représenter tous les comportements possibles du système et tous les états que le système peut atteindre. Nous donnons une procédure de

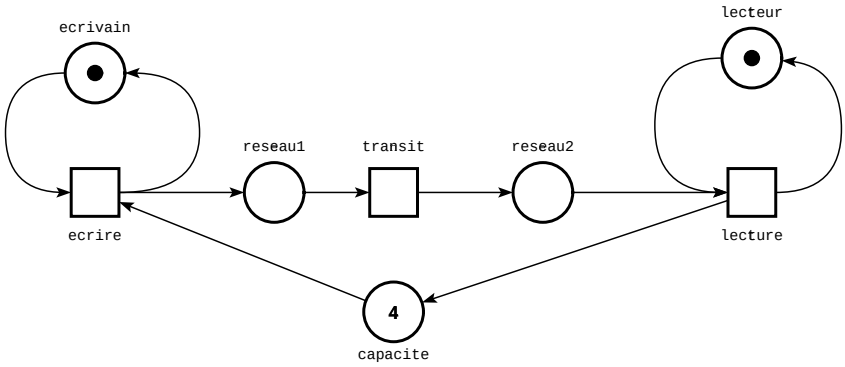


Figure I.4: Écrivain/Lecteur avec contrôle de flux (structurellement borné)

construction d'un tel graphe. Malheureusement la taille du graphe construit explose souvent exponentiellement par rapport à la taille du système. Une partie de notre travail présentée dans la suite a été de proposer des constructions différentes permettant de réduire la taille du graphe construit.

La procédure standard de construction du graphe d'accessibilité est donnée Table I.1 : Tmp contient l'ensemble des états à explorer, S contient l'ensemble des états déjà visités (états accessibles). G contient le STE, i.e. les triplets (s_1, t, s_2) représentant l'arc allant de s_1 à s_2 par la transition t.

Tant qu'il y a des états à explorer on prend l'un d'entre eux (noté s) et on appelle la fonction *develop_exhaustive* sur l'ensemble des transitions sensibilisées dans le marquage correspondant à l'état s. Pour chacune de ces transitions la fonction *develop_exhaustive* calcule le marquage atteint par le tir de cette transition. Cette transition est ajoutée au graphe, de plus si le marquage atteint est visité pour la première fois il est ajouté à l'ensemble S et à l'ensemble Tmp pour explorer ses successeurs ultérieurement.

$Tmp \leftarrow s_0$
$S \leftarrow \{s_0\}$
$G \leftarrow \emptyset$ /* Graphe calculé */
while $NonVide(Tmp)$ **do**
  $s \leftarrow extraire(Tmp)$
  develop_exhaustive(Enabled(s))

develop_exhaustive(E):
  for each t **in** E **do**
    $s' \leftarrow fire(s,t)$
    $G \leftarrow G \cup \{< s, t, s' >\}$
    if $s' \notin S$ **then**
      $S \leftarrow S \cup \{s'\}$
      $Ajouter(Tmp, s')$

Table I.1: Algorithme général de construction du graphe d'accessibilité

La politique de gestion de l'ensemble Tmp influence l'ordre de construction du graphe : une politique FIFO (Tmp est une file) correspond à un parcours en largeur d'abord, une politique LIFO (Tmp est une pile) correspond un parcours en profondeur d'abord. Toute autre politique (et même l'absence de politique !) est également envisageable, nous verrons dans la suite, un exemple de stratégie pour optimiser (de manière heuristique !) la recherche de transitions mortes.

1.3 Réseau de Petri temporel

Un système temps réel est caractérisé par des contraintes temporelles à respecter. Spécifier le système sans tenir compte des contraintes temporelles, revient à spécifier un système plus général (i.e. moins contraint) que le système réel, i.e. un système qui a plus de comportements. Certaines propriétés (comme l'absence de blocage, l'inaccessibilité d'un marquage,...) peuvent être vérifiées sur ce système plus général, en permettant de conclure que le système réel satisfait lui aussi la propriété. Lorsque la propriété ne peut pas être vérifiée sur un sur-comportement ou que le sur-comportement ne satisfait pas la propriété, on ne peut rien conclure sur le système réel. Il est alors nécessaire d'affiner la représentation du système avec un formalisme de description qui permette de spécifier les contraintes temporelles.

Différents modèles temporisés des réseaux de Petri ont été définis (intervalle de temps sur les places, sur les transitions...). Nous nous intéressons plus particulièrement aux réseaux de Petri temporels définis dans [MF76], qui ont des intervalles sur les transitions. Le graphe de classes permet l'analyse du comportement temporel d'un système décrit dans ce formalisme.

Définition 13 *Réseau de Petri temporel*
Un réseau de Petri temporel est un quintuplet $R = \langle P, T, Pre, Post, m_0, I_s \rangle$ *dans lequel* $(\langle P, T, Pre, Post \rangle, m_0)$ *est un réseau de Petri marqué et* I_s *est une fonction* **intervalle statique** *qui associe un intervalle à chaque transition :* $I_s : T \mapsto \mathbb{Q}^+ \times \mathbb{Q}^+ \cup \{\infty\}$.

La fonction I_s associe à chaque transition un intervalle à bornes rationnelles. $Min(I_s(t))$ et $Max(I_s(t))$ désignent les bornes inférieures et supérieures de l'intervalle. Si $I_s(t)$ est non supérieurement borné, on posera $Max(I_s(t)) = \infty$.

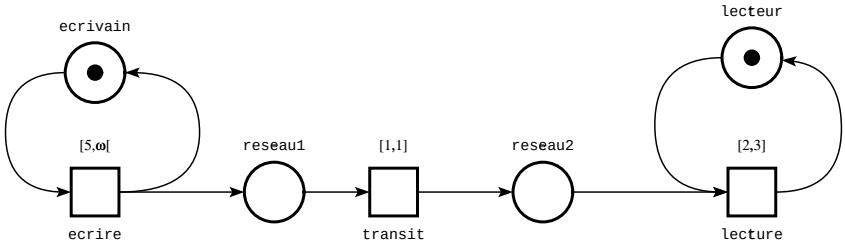


Figure I.5: Ecrivain/Lecteur avec temporisations

La figure I.5 donne un exemple de réseau de Petri temporel, sur un exemple standard d'écrivain/lecteur. Écrire prend au minimum 5 unités de temps. Le transit de l'information prend exactement 1 unité de temps. La lecture prend entre 2 et 3 unités de temps.

1.3.1 État d'un réseau de Petri temporel

Dans le cas d'un réseau de Petri atemporel, il n'est pas très difficile de définir l'état d'un système, en effet celui-ci est caractérisé par le marquage du réseau. Dans le cas d'un système temporel le marquage n'est plus suffisant. On lui associe donc une information temporelle représentée par un intervalle dynamique qui représente dans quel intervalle chaque transition sensibilisée peut-être tirée.

Définition 14 *État d'un réseau temporel*
Un état d'un réseau temporel peut être défini comme un marquage m associé à une fonction I, qui associe un intervalle réel non négatif à chaque transition sensibilisée par m : $I : T \mapsto \mathbb{R}^+ \times \mathbb{R}^+ \cup \{\infty\}$.

Initialement $s_0 = (m_0, I_0)$ avec $I_0(t) = I_s(t)$ pour chaque transition sensibilisée par le marquage initial m_0.

1.3.2 Sémantique des réseaux de Petri temporels

Définition 15 *Sémantique des réseaux de Petri temporels*
Une transition sensibilisée par un marquage m peut être tirée qu'après $Min(I_s(t))$ et doit l'être avant $Max(I_s(t))$ sauf si le tir d'une autre transition désensibilise la transition t avant que celle-ci ne soit tirée. Le tir des transitions est de durée nulle.

Un système peut évoluer, i.e. changer d'état, selon deux mécanismes : soit le tir d'une transition du réseau de Petri, soit l'avancement du temps. Dans un état $s = (m, I)$ caractérisé par le marquage m et la fonction intervalle I :

- une transition $t \in T$ est tirable si elle est sensibilisée par le marquage m et si elle est sensibilisée depuis au moins $Min(I_s(t))$
- le temps peut progresser d'une valeur $\theta \in \mathbb{R}^{*+}$ si cela ne fait pas dépasser le temps de sensibilisation maximal d'une transition sensibilisée : $\forall t \in Enabled(s), \theta \leq Max(I(t))$

On adapte la définition donnée dans [BBD01] pour séparer le franchissement d'une transition discrète et le franchissement d'une transition temps.

Définition 16 *Franchissement d'une transition discrète*
Soit (R, I_s) un réseau de Petri temporel et (m, I) un état. Après le franchissement d'une transition $t \in T$, l'état (m', I') atteint est caractérisé par :
Le nouveau marquage m' est déterminé comme pour le réseau de Petri sous-jacent non temporisé : $\forall p, m'(p) = m(p) - Pre(p,t) + Post(p,t)$.
Pour tout transition k, l'intervalle de tir $I'(k)$ est défini par :

- *Si k est non sensibilisée par m', alors $I'(k)$ est vide.*
- *Si k est distincte de t, sensibilisée par m et n'est pas en conflit avec t ($^\bullet t \cap {}^\bullet k = \emptyset$), alors $I'(k) = I(k)$*
- *Sinon $I'(k) = I_s(k)$.*

Définition 17 *Franchissement d'une transition temps*
Soit (R, I_s) un réseau de Petri temporel et (m, I) un état. Après le franchissement d'une transition $\theta \in \mathbb{R}^{*+}$, l'état (m, I') atteint est caractérisé par la définition de $I'(k)$ pour toute transition discrète k :

– *Si k est sensibilisée par m, alors*

$$I'(k) = \begin{cases} [max(0, Min(I(k)) - \theta), Max(I(k)) - \theta] \text{ si } Max(I(k)) \text{ est fini,} \\ [max(0, Min(I(k) - \theta), \infty[\text{ sinon} \end{cases}$$

– *Sinon $I'(k)$ est vide.*

Tandis que le franchissement d'une transition est discret, l'évolution du temps est continue. Ainsi le temps peut progresser d'une infinité de manières différentes.

Remarquons qu'un intervalle $[0, \infty[$ reste identique lorsque le temps progresse, par conséquent on reste dans le même état.

Cette règle définit sur l'ensemble des états une relation d'accessibilité temporisée notée $\xrightarrow{t}$ ou $\xrightarrow{\theta}$. On a $s \xrightarrow{t} s'$ si la transition t peut être tirée depuis l´état s et $s \xrightarrow{\theta} s'$ si le temps θ peut s'écouler depuis l´état s.

L'ensemble des états d'un réseau temporel est l'ensemble des états accessibles depuis l´état initial s_0, muni de la relation d'accessibilité temporisée $\xrightarrow{t/\theta}$.

Définition 18 *Graphe d'accessibilité*

On définit le graphe d'accessibilité d'un réseau de Petri temporel (R, I_s) par le système de transitions étiquetées $STE(R, I_s) = \langle S, s_0, (T \cup \mathbb{R}^*), \rightarrow \rangle$ dont les états sont les états accessibles et dont les arcs, étiquetés par les noms des transitions ou une information temporelle, sont définis par la relation de tir : $\rightarrow \subseteq S \times (T \cup \mathbb{R}^*) \times S$ avec

$$\forall s_1, s_2 \in S, \forall t \in T, (s_1, t, s_2) \in \rightarrow \text{ ssi } s_1 \xrightarrow{t} s_2$$

et

$$\forall s_1, s_2 \in S, \forall \theta \in \mathbb{R}^*, (s_1, \theta, s_2) \in \rightarrow \text{ ssi } s_1 \xrightarrow{\theta} s_2$$

Les transitions de progression du temps pouvant prendre toute valeur réelle dés que le temps peut progresser, un état admet en général une infinité de successeurs. Par conséquent le graphe d'accessibilité est généralement infini.

1.3.3 Divergence temporelle et blocage

État de divergence temporelle Une divergence temporelle signifie que le temps peut progresser indéfiniment. Donc un état est un « état de divergence temporelle » s'il existe une séquence infinie ne contenant aucune transition discrète. Un état de divergence temporelle est facilement identifiable, puisque son marquage ne sensibilise que des transitions qui ne sont pas supérieurement bornées ($\forall t \in Enabled(m, I), Max(I_s(t)) = \infty$).

Notons qu'une transition atemporelle n'est pas équivalente à une transition temporelle $[0; \infty[$. En effet dans un réseau de Petri atemporel on considère implicitement que le système va évoluer en permanence (hypothèse de progrès : si une transition est sensibilisée en un état alors on ne restera pas indéfiniment dans cet état). Par exemple dans le réseau de Petri de gauche de la figure I.6 l'état $p1$ est obligatoirement atteint, donc la formule $\phi = <> p1$ est vraie. Par contre dans le réseau de Petri temporel de la figure de droite, l'état initial est un état de divergence temporelle, par conséquent on va éventuellement rester indéfiniment dans l'état initial, donc la propriété ϕ est fausse.

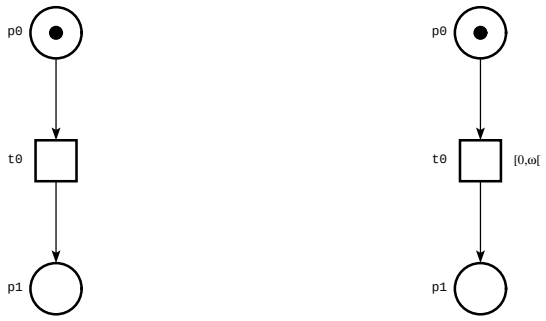


Figure I.6: Réseau de Petri / Réseau de Petri temporel

Blocage Si la notion de blocage est triviale dans un système atemporel, elle est plus délicate à définir pour les systèmes temporisés. Dans le cas des réseaux de Petri temporels, nous retenons la définition suivante : un blocage est un état qui ne permet plus de franchir de transition discrète. Il permet par contre de laisser s'écouler le temps et correspond donc à un état de divergence temporelle.

La situation est différente dans le cas des automates temporisés : un invariant d'état peut par exemple empêcher une progression infinie du temps. Ainsi un état de blocage peut devenir un état d'inconsistance temporelle si un invariant temporel interdit de rester infiniment dans l'état.

Dans les automates temporisés on peut également avoir un état qui devient un état de blocage si on laisse le temps trop progresser. Considérons par exemple l'automate temporisé de la figure I.7 : dans l'état initial la transition t_1 peut être tirée dès que $x > 2$. Mais dès que $x \geq 10$ la garde de la transition t_2 n'est plus vérifiée et par conséquent n'est plus franchissable. Le système se retrouve donc bloqué dans l'état e_2. Cet état devient donc un état de blocage lorsque le temps a « trop » progressé.

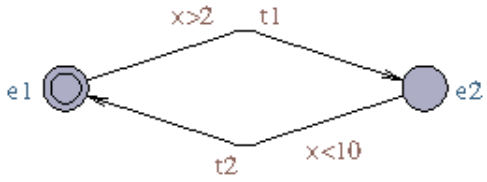


Figure I.7: Exemple d'un état de divergence

2 Modèles de propriétés

Les propriétés peuvent être classées en plusieurs catégories en fonction de la nature de la propriété. Dans la suite nous nous intéressons à deux classes : propriétés générales et propriétés spécifiques.

2.1 Propriétés générales

Cette première partie s'inspire de [HV01]. Les propriétés sont définies dans le cadre du formalisme des réseaux de Petri, mais elles peuvent être adaptées à d'autres formalismes.

Définition 19 *Séquence infinie*
Un réseau de Petri (R, m_0) admet une séquence infinie $w \in T^\omega$ si pour tout w' préfixe fini de w, w' est une séquence d'exécution de (R, m_0).

Un réseau ne possédant pas de séquence d'exécution infinie, vérifie la propriété de **terminaison**.

Cette propriété de terminaison est très importante pour les systèmes dont on veut avoir la garantie qu'ils finissent. Inversement pour certains systèmes réactifs concurrents, on va vouloir vérifier la propriété duale, qui consiste à vérifier que le programme ne s'arrête jamais. Cette propriété est par exemple très importante en avionique pour un calculateur de vol, ou un protocole de communication. En effet la plupart des systèmes ont un fonctionnement infini qui ne doit pas s'interrompre, par conséquent ils ne doivent pas atteindre une situation de blocage dans laquelle plus aucune action n'est possible. Cette propriété est appelée propriété d'absence de blocage ou pseudo-vivacité.

Définition 20 *Pseudo-vivacité*
Un réseau de Petri (R, m_0) est dit **pseudo-vivant** *si :*

$$\forall m \in S(R, m_0), \exists t \in T \text{ tel que } m \overset{t}{\rightarrow\!\!\!\!\rightarrow}$$

Définition 21 *État de blocage*
Un marquage qui ne sensibilise aucune transition est un marquage mort ou état de blocage (deadlock state).

$$m \in S(R, m_0) \text{ est un état de blocage ssi } \forall t \in T, m \overset{t}{\not\rightarrow}$$

Vérifier cette propriété sur le graphe d'accessibilité revient simplement à vérifier s'il existe un état qui n'a pas de successeur. On peut également observer que cette propriété peut très simplement être vérifiée «au vol» («on the fly»), i.e. sans construire en totalité le graphe d'accessibilité avant de commencer la vérification, mais au contraire en construisant et vérifiant simultanément. En cas de violation, preuve de blocage, on peut arrêter la construction. Pour cela l'algorithme est complété pour tester dans chaque état atteint si l'ensemble des transitions sensibilisées est vide. L'algorithme modifié est donné dans la table I.2. Cet algorithme utilise la même fonction d'exploration : *develop_exhaustive*.

Dans le cas où un blocage existe, une vérification au vol permet de le trouver plus rapidement, mais dans le cas où le système ne contient pas d'état de blocage, une vérification au vol ne permet pas d'accélérer le processus.

Une autre information intéressante est la présence de transitions «inutiles». En effet, dans la logique d'une conception, le concepteur ne définit que les transitions qui lui semblent nécessaires au fonctionnement du système, il ne rajoute pas des transitions qui représentent des évènements « sans but ». Aussi il est souvent très intéressant de vérifier que chaque transition du réseau est effectivement franchissable. Une transition n'ayant aucune occurrence est dite **transition morte**. De même en programmation une partie de code qui n'est jamais exécutée s'appelle « code mort ». Du code mort est souvent dû à une erreur de conception ou un manque de compréhension du système. Aussi détecter et interdire le code mort est une méthode souvent utilisée pour améliorer

```
Pile ← s_0
S ← {s_0}
G ← ∅ /* Graphe calculé */
while NotEmpty(Pile) do
   s ← pop(Pile)
   if Enabled(s) = ∅ then
     print "Deadlock"
   else
      develop_exhaustive(Enabled(s))
```

Table I.2: Algorithme en profondeur d'abord de recherche au vol des états de blocage

la qualité du code. Par exemple dans le cas des certifications de niveau élevé, une couverture du code par des tests est exigée à cent pour cent pour traquer et éliminer le code mort.

A l'opposé, il est possible de « coder en dur » certaines propriétés en ajoutant au système un « observateur ». Vérifier que le système satisfait la propriété correspondra par exemple à s'assurer que la transition dite « d'observation » n'est jamais possible.

Définition 22 *Transition morte*
Une transition t d'un réseau de Petri (R, m_0) est morte si et seulement si

$$\forall m \in S(R, m_0), m \overset{t}{\not\gg}$$

Définition 23 *Réseau quasi-vivant*
Un réseau de Petri (R, m_0) est **quasi-vivant** *si aucune transition n'est morte.*

$$\forall t, \exists m \in S(R, m_0) \text{ tel que } m \overset{t}{\gg}$$

Une exemple de vérification de cette propriété sur un exemple de taille importante est donné dans le chapitre III. Nous verrons en particulier qu'on peut utiliser une heuristique d'exploration particulière pour aider la vérification de cette propriété.

Définition 24 *Réseau vivant*
Un réseau de Petri (R, m_0) est **vivant** *si et seulement si :*

$$\forall m \in S(R, m_0), \forall t \in T, \exists m' \in S(R, m) \text{ tel que } m' \overset{t}{\gg}$$

La vivacité garantit non seulement que le système est quasi-vivant, donc avec aucune transition morte, mais qu'en plus toutes les transitions du système peuvent redevenir actives depuis n'importe quel état.

Pour les systèmes cycliques, en plus de savoir que toutes les actions restent toujours possibles dans le futur, il est souvent important de savoir si le système peut toujours revenir dans son état initial. On introduit donc la notion de réseau de Petri réinitialisable.

Définition 25 *Réseau réinitialisable*
Le réseau de Petri (R, m_0) est réinitialisable si et seulement si

$$\forall m \in S(R, m_0), m_0 \in S(R, m)$$

On obtient de façon très évidente la propriété suivante :

Propriété 2 *Un réseau de Petri quasi-vivant et réinitialisable est vivant.*

2.2 Propriétés spécifiques : Logique Temporelle Linéaire

Différents formalismes permettent d'exprimer des propriétés qui doivent être satisfaites par le système. Même si l'expressivité d'un formalisme est une caractéristique très importante, elle n'est pas la seule à prendre en compte. Il faut également être capable de décrire une procédure de vérification permettant d'établir la satisfaction de la propriété par le système. La complexité en temps et en espace de cette procédure sont également des caractéristiques importantes. Enfin on peut aussi s'intéresser à la facilité d'apprentissage et d'utilisation du formalisme.

Considérant ces nombreux paramètres nous nous sommes plus particulièrement intéressés à la logique temporelle linéaire, qui présente un pouvoir d'expression déjà très intéressant, tout en permettant une vérification efficace que nous avons pu implémenter. La logique LTL permet d'exprimer des propriétés spécifiques sur les séquences d'exécution du système. Dans la suite nous commençons par définir la logique LTL puis donner des exemples d'utilisation.

La logique temporelle linéaire est constituée des opérateurs de la logique classique (et, ou, non) et de deux modalités « Until » et « next-time ». Intuitivement la modalité « Until » notée $a\mathcal{U}b$ permet d'exprimer que a va rester vrai jusque à l'apparition de b (et b arrivera obligatoirement). La modalité « next-time » notée $\mathcal{X}a$ signifie que a est vrai dans le prochain état.

Définition 26 *Syntaxe de LTL*

Étant donné un ensemble de propositions atomiques AP et $p \in AP$. Les formules LTL sont définies par la grammaire suivante :

$$\Phi ::== p \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \mathcal{U} \Phi_2 \mid \mathcal{X}\Phi_1$$

Ce qui signifie :

1. les propositions atomiques $p \in AP$ sont des formules

2. si Φ_1 et Φ_2 sont des formules alors $\neg\Phi_1$, $\Phi_1 \vee \Phi_2$, $\Phi_1 \mathcal{U} \Phi_2$ et $\mathcal{X}\Phi_1$ sont des formules.

3. toute expression obtenue en appliquant un nombre fini de fois les deux règles précédentes est une formule.

Une formule LTL s'évalue sur une séquence de propositions atomiques.

Définition 27 *Sémantique de LTL*

Soit AP un ensemble de propositions atomiques et $\sigma = s_0.s_1 \ldots$ une séquence d'états avec $\nu : S \rightarrow 2^{AP}$ qui associe à chaque état s_i l'ensemble $\nu(s_i)$ des propositions atomiques vérifiées en s_i.

On note $(\sigma, i) \models \Phi$ si la séquence σ satisfait la formule Φ à la position i.

$(\sigma, i) \models p$ *ssi* $p \in \nu(s_i)$, *avec* $p \in AP$

$(\sigma, i) \models \neg\Phi$ *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* $\exists j \geq i$ *tel que* $(\sigma, j) \models \Phi_2$ *et* $\forall k, i \leq k < j$, $(\sigma, k) \models \Phi_1$

$(\sigma, i) \models \mathcal{X}\Phi$ *ssi* $(\sigma, i+1) \models \Phi$

On note $\sigma \models \Phi$ pour $(\sigma, 0) \models \Phi$.

La sémantique de LTL est définie pour les séquences infinies. Dans notre cas, nos systèmes peuvent aussi avoir des séquences maximales finies en cas d'état de blocage, par conséquent nous étendons la sémantique aux séquences finies.

Définition 28 *Sémantique LTL sur une séquence finie*

Soit $\sigma = s_0.s_1 \ldots .s_n$ une séquence finie.

$$\sigma \models \phi \text{ ssi } s_0.s_1 \ldots .s_n.s_n.s_n \ldots \models \phi$$

Après avoir défini la satisfaction d'une formule par une séquence d'exécution, on définit la satisfaction d'une formule par un système.

Définition 29 *Satisfaction d'une formule par un système*
Une structure de Kripke $\mathcal{SK}$ satisfait la formule Φ, ssi pour toute séquence d'exécution σ partant de l'état initial de $\mathcal{SK}$, $\sigma \models \Phi$. On note alors $\mathcal{SK} \models \Phi$

Les opérateurs suivants sont couramment définis :
- Et : $\phi \wedge \psi \stackrel{Def}{=} \neg(\neg\phi \vee \neg\psi)$
- Implique : $\phi \Rightarrow \psi \stackrel{Def}{=} \neg\phi \vee \psi$
- Relache : $\phi \mathcal{R} \psi \stackrel{Def}{=} \neg(\neg\phi \mathcal{U} \neg\psi)$
- Inévitablement : $<> \phi \stackrel{Def}{=} true \mathcal{U} \phi$
- Toujours : $[] \phi \stackrel{Def}{=} false \mathcal{R} \phi = \neg <> (\neg\phi)$

Ces opérateurs n'augmentent pas le pouvoir d'expression de la logique, mais sont très utiles car ils permettent de simplifier l'écriture de certaines formules.

Exemples de formules qui sont vérifiées dans l'exemple de la Figure I.4 :
- La place $reseau1$ est marquée infiniment souvent :

$$\varphi_1 = [] <> reseau1$$

- L'envoi d'un message sur $reseau1$ est inévitablement suivie par l'envoi d'un message sur $reseau2$:

$$\varphi_2 = [] (reseau1 \Rightarrow <> reseau2)$$

Notons que la propriété φ_2 n'est pas vérifiée dans le système de la Figure I.3 qui était non borné. En effet la séquence infinie $ecrire.ecrire.ecrire...$ entraîne la violation de la formule.

3 Vérification d'une propriété LTL

Un système satisfait une formule logique LTL, si toutes les séquences d'exécution du système satisfont la formule. Les procédures de « model checking » pour les propriétés linéaires sont basées sur la recherche d'un contre-exemple, i.e. un comportement du système qui ne satisfait pas la formule. Puisque l'analyse est exhaustive, on peut assurer que l'absence de contre-exemple assure la satisfaction de la propriété par le système.

Pour chercher un comportement qui viole la formule à vérifier, on construit d'abord un automate de Büchi représentant la négation de la formule. A partir de cet automate de Büchi et de l'automate du système, on construit un automate de Büchi synchronisé dont les séquences acceptantes sont des comportements possibles du système qui satisfont la négation de la formule. La recherche exhaustive d'un contre-exemple consiste alors à vérifier s'il existe une séquence acceptante pour cet automate synchronisé.

Le schéma de la figure I.8 illustre les différentes étapes du « model checking » d'une formule LTL sur un système. Chacune de ces étapes est détaillée dans les sections suivantes.

Dans le cas général, vérifier si un système satisfait une formule de LTL est PSPACE-complet [Val98]. L'algorithme donné ici par la construction d'un automate de Büchi et d'un graphe synchronisé est linéaire par rapport à la taille du graphe de comportement du système et **exponentiel par rapport au nombre d'états de l'automate de Büchi**. Par conséquent cette approche sera utilisable en pratique à la condition que le nombre d'états de l'automate de Büchi soit relativement faible.

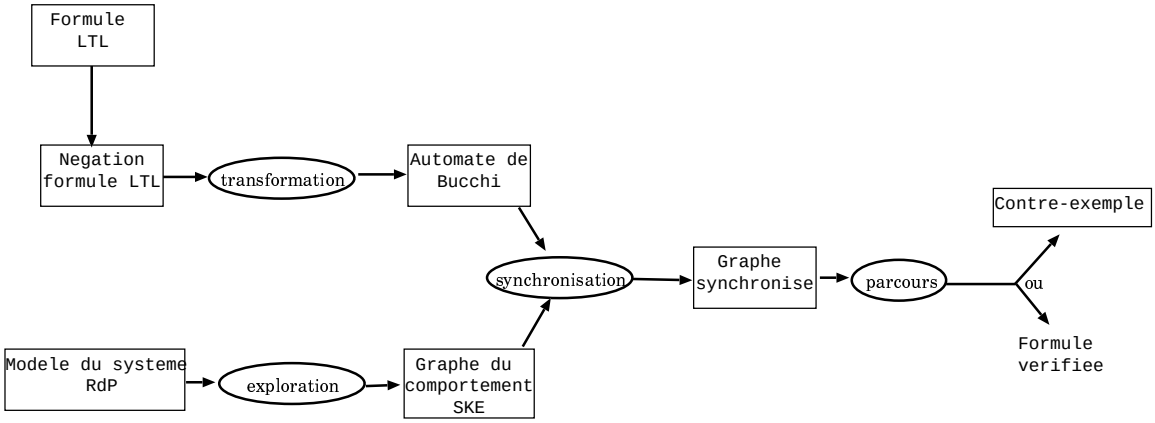


Figure I.8: Model checking d'une formule LTL

3.1 Caractérisation des états de blocage

Un traitement particulier est fait pour les états qui n'ont aucune transition du système sensibilisée (i.e. les états de blocage du système) :

- D'une part, une ϵ transition de l'état dans lui même est ajoutée enfin de rendre toute séquence, arrivant dans cet état, infinie, puisque les propriétés LTL sont définies sur des séquences infinies.
- D'autre part, une variable propositionnelle « deadlock » est ajoutée à cet état.

La variable propositionnelle « deadlock » permet d'exprimer des formules LTL qui permettent de caractériser plus finement les états de blocage.

Le blocage est inéluctable :

$$<> \ deadlock$$

L'état initial est un état bloquant (caractéristique courante d'une mauvaise initialisation du système) :

$$[] \ deadlock$$

Il n'y a pas d'état de blocage dans le comportement du système :

$$[] \ (\ \neg \ deadlock \ )$$

3.2 Différentes étapes de la vérification

3.2.1 Construction d'un automate de Büchi correspondant à une formule LTL

Définition 30 *Automate de Büchi*
Un automate de Büchi est un quintuplet $\mathcal{A}_{Büchi} = \langle AP, S, \{\overset{e}{\longrightarrow}\}_{e \in 2^{AP}}, s_0, Acc \rangle$ *défini par :*

- AP *est un ensemble de propositions atomiques*
- S *est un ensemble d'états*
- $\overset{e}{\longrightarrow}$ *est une relation incluse dans* $S \times 2^{AP} \times S$
- s_0 *un état particulier de* S *(état initial)*
- $Acc \subseteq S$ *ensemble d'états d'acceptants*

Pour définir le langage de l'automate de Büchi on définit les séquences qui sont acceptées par celui-ci.

Définition 31 *Séquence acceptée par l'automate de Büchi*

Soit $\sigma = x_0x_1x_2\ldots \in (2^{AP})^\omega$ *une séquence infinie, avec* $s_0 \overset{x_0}{\twoheadrightarrow} s_1 \overset{x_1}{\twoheadrightarrow} s_2\ldots$
σ *est acceptée par l'automate de Büchi ssi* $\forall i\ \exists j$ *tel que* $j \geq i$ *et* $s_j \in Acc$.

Étant donnée une formule Φ, on souhaite construire un automate de Büchi tel que toute séquence σ telle que $\sigma \models \neg\Phi$ soit une séquence acceptante pour l'automate de Büchi.

Considérons par exemple la formule $\Phi = [](t1 \Rightarrow \mathcal{X}(<> t2))$ exprimant qu'une action $t1$ est inévitablement suivie par une action $t2$ dans le futur. Soit $\Psi = \neg\Phi$ la négation de la formule. Les séquences acceptantes de l'automate de Büchi de la figure I.9 satisfont la formule Ψ. En effet une séquence contre-exemple de la formule Φ sera une séquence pour laquelle une action $t1$ ne sera jamais suivie de l'action $t2$ (La négation est notée par $-t2$).

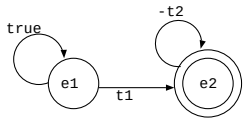


Figure I.9: Automate de Büchi de la formule $\Psi = \neg[](t1 \Rightarrow \mathcal{X}(<> t2))$

Pour montrer l'importance de passer une infinité de fois par un état acceptant nous allons considérer une nouvelle formule. Supposons que l'on veuille vérifier que le système ne puisse pas faire une infinité de fois l'action $t1$, on va donc vérifier la propriété $\Phi = \neg[] <> t1$. Un contre exemple sera donc une séquence infinie qui contient un nombre infini de fois l'action $t1$. Un automate de Büchi reconnaissant ces mots est donné dans la figure I.10 : l'état $e2$ est un état acceptant, passer une fois par l'état $e2$ ne suffit pas pour être un contre-exemple, il faut une infinité de $t1$, donc passer une infinité de fois pas l'état $e2$.

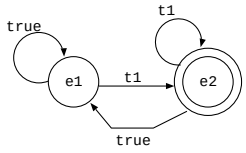


Figure I.10: Automate de Büchi de la formule : $\Psi = [] <> t1$

Pour une même formule Φ plusieurs automates de Büchi peuvent être construits, or la taille de l'automate a une influence très importante sur la taille du graphe synchronisé. Il est donc important d'apporter une attention toute particulière à cette construction. Des techniques ont été développées pour construire un automate le plus petit possible ou pour réduire la taille de l'automate construit [GO01].

3.2.2 Synchronisation du graphe d'accessibilité et de l'automate de Büchi

Définition 32 *Produit synchronisé*

Soit une structure de Kripke étiquetée $\mathcal{SKE} = \langle AP_C, T_C, S_C, init_C, \{\overset{t}{\longrightarrow}\}_{t\in T_C}, \nu\rangle$ *représentant le comportement du système et* $B = \langle AP_B, S_B, \{\overset{exp}{\longrightarrow}\}_{exp\in 2^{AP}}, init_B, Acc_B\rangle$ *l'automate de Büchi correspondant à la négation de la formule.*

Le produit synchronisé est défini par l'automate de Büchi $\langle AP, S, \{\overset{(t,e)}{\longrightarrow}\}_{(t,e)\in T_C\times 2^{AP}}, s_0, Acc\rangle$ *avec*

- $AP = AP_C \cup AP_B$ *est un ensemble de propositions atomiques*
- $S \subseteq S_C \times S_B$ *est un ensemble d'états*
- $\rightarrow \subseteq S \times S$ *est défini par :* $\forall s, s' \in S$ *tels que* $s = (s_C, s_B)$ *et* $s' = (s'_C, s'_B)$, $s \xrightarrow{(e,t)} s'$ *ssi* $[\, s_C \xrightarrow{t} s'_C$ *et* $s_B \overset{e}{\twoheadrightarrow} s'_B$ *et* $e \in \nu(s_C)\,]$
- $s_0 = (init_C, init_B)$ *un état particulier de* S *(état initial)*
- $Acc = S_C \times Acc_B$ *ensemble d'états d'acceptation*

3.2.3 Recherche d'une séquence contre-exemple

Après avoir construit le produit synchronisé, on recherche une séquence infinie acceptante pour l'automate de Büchi synchronisé. Une telle séquence est un contre exemple, montrant que la propriété n'est pas vérifiée, puisqu'elle est à la fois un comportement du système et une négation de la formule.

Les séquences acceptantes sont des séquences infinies qui passent une infinité de fois par un état acceptant pour le système et une infinité de fois par un état acceptant de l'automate de Büchi de la négation de la formule. Comme par définition, tout état de l'automate du système est un état acceptant, il suffit de trouver une séquence qui passe une infinité de fois par un état acceptant pour l'automate de Büchi de la négation de la formule.

Chercher une telle séquence consiste à chercher une composante fortement connexe dans le graphe synchronisé, qui comporte au moins un état acceptant de l'automate de Büchi. Dans ce cas on conclut que le système ne satisfait pas la formule, dans le cas contraire il la satisfait.

3.3 Vérification de propriétés sur des systèmes temporels

Dans la pratique le graphe d'accessibilité ne peut généralement pas être construit puisqu'il est infini. Une abstraction du comportement est donc nécessaire pour regrouper un nombre infini d'états dans une classe. Il est important que l'abstraction construite préserve la propriété que l'on souhaite vérifier. Différentes constructions ont été proposées pour différentes classes de propriétés : graphe de classes préservant les propriétés LTL, graphe de classes atomiques[BV03] pour préserver CTL.

3.3.1 Graphe de classes préservant LTL

Le graphe de classes regroupe une infinité d'états dans une classe. La progression du temps est abstraite en considérant qu'une séquence de transitions temps suivie par une transition discrète t peut être abstraite par le tir d'une transition temps égale à la somme des temps de la séquence suivie immédiatement par la transition discrète. Cette abstraction occulte la possibilité de divergence temporelle ou de façon équivalente repose sur une hypothèse de progression : si une action discrète est possible alors on ne va pas laisser écouler un temps infini. Cette abstraction ne permet donc pas la représentation d'une séquence infinie de transitions temps, par conséquent nous allons ajouter une transition de la classe dans elle-même lorsque une telle séquence est possible. La construction détaillée du graphe de classes est donnée dans [BD91], nous rappelons seulement une définition possible des classes d'états donnée dans [BRV03a] ; cette définition met en évidence la relation entre les classes et les états qu'elles contiennent.

Définition 33 *Classes d'états*

Les espaces de classes d'états sont des recouvrements de l'ensemble des états munis d'une relation de transition $\stackrel{t}{\rightarrow}$ satisfaisant la propriété suivante :

$$(\forall t \in T)(\forall c, c')(c \stackrel{t}{\rightarrow} c' \Leftrightarrow (\exists s \in c)(\exists s' \in c')(s \stackrel{t}{\rightarrow} s'))$$

où c et c' désignent des classes d'états.

Nous supposerons de plus que tous les états de chaque classe ont même marquage.

Un théorème important de l'article[BD91] concerne la finitude du graphe construit.

Théorème 1 *Caractère borné d'un réseau de Petri temporel*

Si le réseau de Petri atemporel est borné, alors le réseau de Petri temporel est borné et le graphe de classes fini.

Bien que [BD91] donne des conditions nécessaires à la non terminaison de la construction du graphe de classes, dans l'état actuel de la recherche, aucune condition suffisante (permettant de conclure que le graphe de classes est infini) est connue. Par conséquent dès que le réseau de Petri atemporel est non borné on n'a aucune procédure de décision qui permette de garantir la terminaison ou la non terminaison de la construction. Cette limite théorique est souvent résolue dans la pratique en mettant une borne maximale sur le marquage des places, permettant ainsi d'interrompre la construction si le marquage d'une place dépasse cette limite. La connaissance du système modélisé permet généralement de connaître une majoration du nombre maximal de jetons dans une place.

Notons que les transitions entre classes d'états ne portent pas d'information temporelle, les classes d'états permettent d'abstraire le temps du comportement d'un réseau temporel. La progression du temps permet d'atteindre un état qui appartient à la même classe.

En effet $\forall \theta \in \mathbb{R}^+$ tel que $s \stackrel{\theta}{\longrightarrow} s'$, Si $s \in c$ Alors $s' \in c$.

3.3.2 Représentation explicite des états de divergence temporelle

Le graphe de classes préserve les propriétés LTL modulo l'hypothèse de non divergence temporelle, cela signifie que l'on a les mêmes séquences de transitions entre le système réel et le graphe de classes. Cependant pour vérifier une propriété LTL sur un système temporisé, il faut tenir compte des états de divergence. En effet une formule telle que « Inévitablement e1 arrive » $= <> e1$ ne peut pas être vérifiée sur un graphe de comportement où les états de divergence ne sont pas représentés.

Le graphe de classe dans sa version standard permet de distinguer les états de divergence en analysant les contraintes temporelles de la classe. Une classe pour laquelle toutes les transitions sensibilisées t ont $Max(I(t)) = \infty$ est un état de divergence temporelle.

Pour réutiliser les algorithmes d'analyse utilisés pour les systèmes atemporels, on ajoute une transition δ de la classe dans elle-même pour toutes les classes divergentes. La figure I.11 donne un exemple de réseau de Petri temporel et de son graphe de classes auquel on ajoute la transition δ (notée *delta* dans la représentation graphique). Dans la classe initiale, la transition t_1 est tirable dans l'intervalle $[4, w[$ et la transition t_0 dans l'intervalle $[0, w[$ [1], par conséquent cette classe est divergente. De même dans les classes 1 et 3 du graphe, aucune transition n'est sensibilisée, donc on va rester indéfiniment dans la même classe. Notons que dans la classe 2 caractérisée par le

[1] C'est l'intervalle par défaut pour un réseau de Petri temporel, lorsqu'il n'est pas précisé

marquage $p2$, deux transitions ($t2$ et $t3$) sont sensibilisées, mais la transition $t3$ ne sera jamais tirable, puisque la transition $t2$ devra au plus tard être tirée après trois unités de temps, or la transition $t3$ ne serait tirable qu'après quatre unités. Par exemple la propriété LTL $\Phi = <> (\neg p0)$ est fausse puisque on peut rester indéfiniment dans l'état initial. Par exemple la séquence infinie $\delta.\delta.\delta \ldots$ ne contient pas d'état sans le marquage de la place $p0$.

Avec l'ajout de cette transition δ, l'analyse du comportement temporel peut être faite sur ce graphe de classes dénué de toutes les informations temporelles en utilisant le même algorithme (et donc les mêmes logiciels) que la vérification d'un système atemporel.

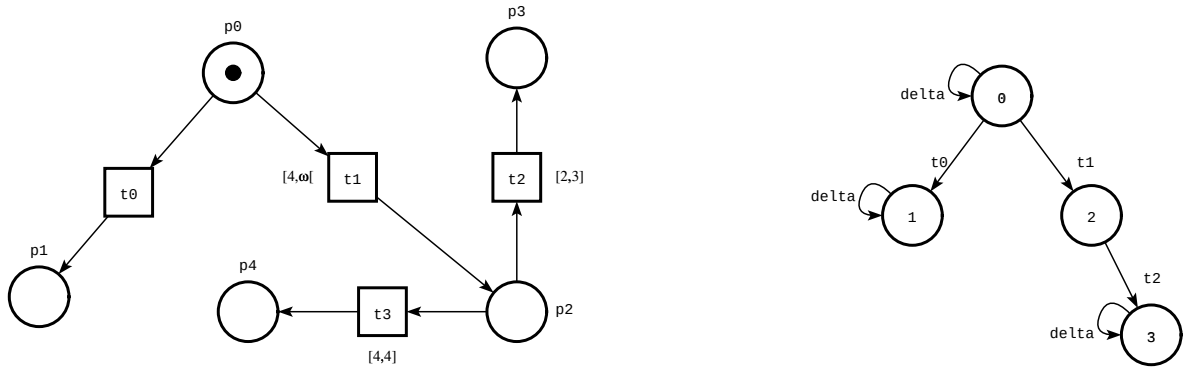


Figure I.11: Exemple d'un état de divergence temporelle

3.4 Implémentation d'un « Model Checker »

3.4.1 Description générale

Le graphe d'accessibilité est construit à l'aide de l'outil *Tina* développé au LAAS, pour lequel nous avons implémenté les algorithmes de réduction de graphe présentés dans les chapitres suivants. L'outil *Tina* propose différents formats pour représenter le graphe d'accessibilité en fonction de l'analyse que l'on souhaite mener dessus. On peut citer en particulier un format textuel, un format « *Aldébaran* » permettant l'utilisation de l'outil de ce même nom et enfin un format décrivant complètement le graphe par un SKE permettant d'obtenir l'étiquetage des transitions et les valuations des états. C'est ce dernier format qui va nous permettre ici de faire le model checking du graphe obtenu.

Nous avons programmé un outil appelé *Mc* permettant de vérifier une propriété LTL sur un graphe de comportement. La phase 1 (construction d'un automate de Büchi acceptant les mots qui ne satisfont pas la formule LTL) est faite en appelant de façon transparente pour l'utilisateur le logiciel *LTL2BA* développé au LIAFA par Paul Gastin et Denis Oddoux[GO01]. L'outil construit alors l'automate de Büchi synchronisé (phase 2) et recherche la présence d'une composante connexe contenant un état acceptant (phase 3). Il permet soit d'établir la satisfaction de la formule, soit de donner une séquence contre-exemple à la formule.

La séquence contre-exemple d'une formule peut être relativement longue (en particulier lors de la vérification d'un système temporisé). Il est alors difficile pour le modélisateur de l'analyser et l'interpréter. Nous avons travaillé à l'affichage d'un contre-exemple sous une forme abrégée : une transition de la séquence contre-exemple n'est donnée que si elle fait changer d'état l'automate de Büchi. En effet, les différents états de l'automate de Büchi représentent les différentes étapes de

la démonstration que la séquence est un contre-exemple. L'affichage de ces instants particuliers permet donc généralement de comprendre pourquoi le système viole la propriété.

Le model checker que nous avons implémenté est basé sur la sémantique $SE - LTL$[CCO+04] qui permet d'exprimer des propriétés LTL en utilisant à la fois des propriétés d'état et des transitions. De plus pour prendre en compte la spécificité des réseaux place/transition, une propriété d'état est valuée. Ceci permet d'exprimer par exemple la propriété suivante : si l'action a_1 a lieu alors on arrivera inévitablement dans un état avec la valuation de la proposition atomique p_1 qui sera exactement égale à 5.

$$[](a_1 \Rightarrow <> (p_1 = 5))$$

Dans le cas des réseaux de Petri, la valuation d'une proposition atomique est égale au nombre de jetons dans la place. Une proposition atomique p_1 est équivalente à $p_1 \geq 1$.

La chaîne d'outils disponible est décrite dans la figure I.12. Elle permet de vérifier une propriété LTL sur un système modélisé par un réseau de Petri ou un réseau de Petri temporel.

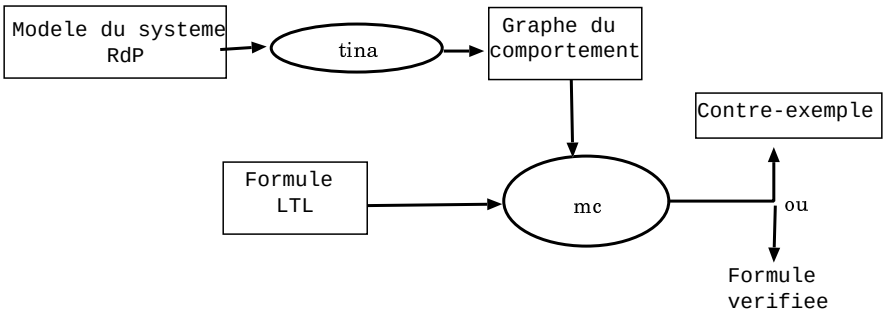


Figure I.12: Implémentation d'un Model checker

3.4.2 Exemple d'utilisation

Dans la version actuelle, l'outil a deux modes de fonctionnement : un mode « batch » et un mode interactif. Pour analyser un système l'outil a besoin du nom des transitions et la valuation de chaque état (i.e. le marquage dans le cas des réseaux de Petri) ; l'outil utilise donc une structure de Kripke étiquetée (compressée dans le format ktz).

Nous donnons ici un petit exemple d'utilisation du model checker sur l'exemple du token ring. L'objectif de ce petit exemple est uniquement de montrer le type de propriétés que l'on peut écrire et l'interface fournie. L'exemple du token ring est décrit en Annexe D.

```
>tina tokenring04.ktz tokenring04.ndr
```

Calcul du graphe d'accessibilité pour l'exemple du jeton tournant.

```
>mc tokenring04.ktz
Mc version 2.7.4 - 03/29/05 - LAAS/CNRS
ktz loaded, 224 states, 688 transitions
0.020s
```

Lancement du model checker en mode Interactif. Chargement du graphe : 224 états et 688 transitions

```
- [] (cs_1 + cs_2 + cs_3 + cs_4 <=1) ;
TRUE
0.050s
```

Vérification de la Section critique : au plus un site a le jeton

```
- infix x suivipar y = [] ( x => () <> y) ;
infix operator suivipar : prop # prop -> prop
0.000s
```

Définition d'un nouvel opérateur infixe : Il est toujours vrai que si x arrive (transition ou état) alors dans le futur y arrivera (transition ou état)

```
- work_1 suivipar release_1 ;
TRUE
0.020s
```

Lorsque la station 1 est dans l'état *work* elle va obligatoirement faire l'action *release*

```
- wait_1 suivipar work_1 ;
FALSE
  state 0 : idle_1 idle_2 idle_3 idle_4 token_in_1
  -transmit_1 ... (preserving T)->
  state 125 : token_out_4 wait_1 wait_2 wait_3 wait_4
   -exit_4 ... (preserving wait_1)->
*  [accepting] state 126 : after_4 wait_1 wait_2 wait_3
wait_4
  -entry_1 ... (preserving - work_1)->
  state 126 : after_4 wait_1 wait_2 wait_3 wait_4
0.060s
```

Un site qui attend finira par travailler : Faux. Depuis l'état initial on peut faire une séquence commençant par $transmit\_1$, puis une séquence qui commence par $exit\_1$ et qui satisfait $wait\_1$. L'étoile représente le début de la boucle. On peut faire une séquence commençant par $entry\_1$ et qui ne contient pas la transition $work\_1$, qui nous ramène au même état que précédemment.

```
- output fullproof ;
output mode set
0.000s
```

Changement de mode : Le contre-exemple est donné entièrement

```
- wait_1 suivipar work_1 ;
FALSE
   state 0 : idle_1 idle_2 idle_3 idle_4 token_in_1
   -transmit_1->
   state 1 : idle_1 idle_2 idle_3 idle_4 token_out_1
   -exit_1->
   state 2 : after_1 idle_1 idle_2 idle_3 idle_4
   -entry_2->
   state 3 : idle_1 idle_2 idle_3 idle_4 token_in_2
   -ask_1->
   state 4 : idle_2 idle_3 idle_4 token_in_2 wait_1
   -transmit_2->
   state 5 : idle_2 idle_3 idle_4 token_out_2 wait_1
   -exit_2->
   state 6 : after_2 idle_2 idle_3 idle_4 wait_1
   -entry_3->
```

La séquence complète contient 140 transitions, elle est donnée en annexe D

Chapter II

Maîtrise de l'explosion combinatoire

1 L'explosion combinatoire

La taille du graphe exhaustif représentant tous les comportements possibles du système est souvent en relation exponentielle avec la taille du système. Ce phénomène bien connu sous le nom d'explosion combinatoire est un facteur très limitant pour l'utilisation industrielle des techniques de vérification formelle. Les causes de l'explosion combinatoire sont diverses : fort parallélisme du système, plages de valeurs importantes pour les variables, contraintes temporelles... Nous allons plus particulièrement nous intéresser à l'explosion combinatoire due à la représentation du parallélisme par l'entrelacement d'actions et à l'explosion combinatoire due aux contraintes temporelles.

1.1 Représentation du parallélisme par l'entrelacement d'actions

Lorsque plusieurs actions sont possibles et peuvent donc être exécutées en parallèle par le système, la sémantique d'entrelacement adoptée explore toutes les combinaisons possibles d'entrelacement de ces actions. Aussi lorsque le système est caractérisé par un niveau de parallélisme important, le nombre d'états à explorer devient trop important. Pour illustrer l'explosion combinatoire, considérons le scheduler de Milner [Mil80].

Exemple 1 *Scheduler de Milner [Mil85]*
n sites exécutent cycliquement l'action A_i puis l'action B_i. Un scheduler contraint l'exécution du système complet de telle sorte que les n sites alternativement exécutent les actions $A_1, A_2, \ldots A_n$. Le réseau de la figure II.1 représente le système complet (scheduler et n sites), tandis que la Figure II.2 donne le graphe d'accessibilité obtenu avec 2 sites.

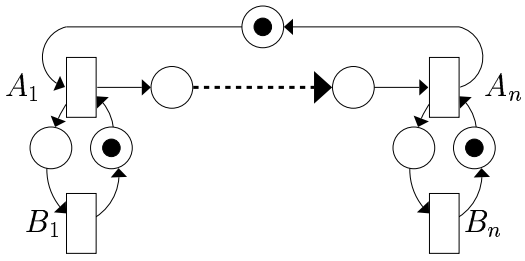


Figure II.1: Scheduler de Milner - Modèle réseau de Petri

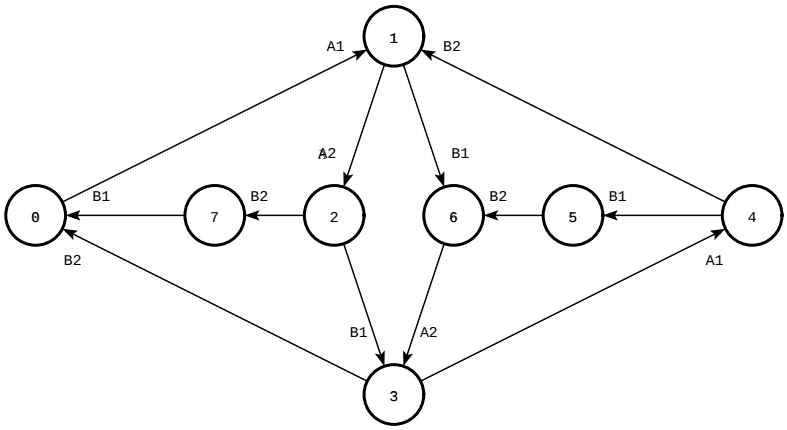


Figure II.2: Scheduler de Milner - Graphe d'accessibilité pour $n = 2$

La Table II.1 et la Figure II.3 donnent le nombre d'états du système exhaustif en fonction du nombre de sites. Intuitivement, cette explosion est due au fait qu'après son action A_i chaque site peut attendre plus ou moins longtemps avant de faire son action B_i. Par conséquent de nombreuses configurations sont possibles. Le calcul analytique permet de montrer que le graphe a $2^n * n$ états. Cette croissance exponentielle de la taille du graphe par rapport à la taille du système est appelée « explosion combinatoire ».

Nombre de sites	2	4	10	15	16
Taille du graphe	8	64	10 240	491 520	1 048 576

Table II.1: Scheduler de Milner - Taille du graphe d'accessibilité à n sites

1.2 Conséquences des contraintes temporelles

L'ajout des contraintes temporelles sur le système limite les comportements possibles. On peut par exemple avoir un comportement atemporel non borné qui devient borné lorsqu'on lui applique des contraintes temporelles. Si on considère un exemple non temporisé du modèle Ecrivain/Lecteur (Figure I.3) le nombre de marquages accessibles est non borné. La version temporisée donnée figure I.5 est bornée : le graphe de classes, représenté figure II.4, est fini, ce qui signifie que le nombre de marquages accessibles en tenant compte des contraintes temporelles est fini.

La limitation des comportements possibles par les contraintes temporelles provoque parfois la construction d'un graphe de classes plus petit que le graphe du comportement atemporel. Cette situation se produit en particulier lorsque de nombreuses contraintes temporelles sont rigides (des intervalles $[d_{min}, d_{max}]$ avec $d_{min} = d_{max}$) et contraignent fortement le système. Cependant l'exemple de la figure II.6 montre que les contraintes rigides ne sont pas une condition suffisante pour éviter l'explosion du nombre de classes.

Considérons par exemple une variante temporisée du planteur de bananes présenté dans [Had01] (et mis dans l'annexe C). La figure II.5 présente cette variante : Le réseau de Petri ne modélise plus une journée mais la vie du planteur de bananes. Une place énergie représente l'énergie du cueilleur de banane ; chaque action consomme plus ou moins d'énergie (poids sur les arcs), les actions dormir et manger lui permettent de récupérer de l'énergie. Si le planteur n'a

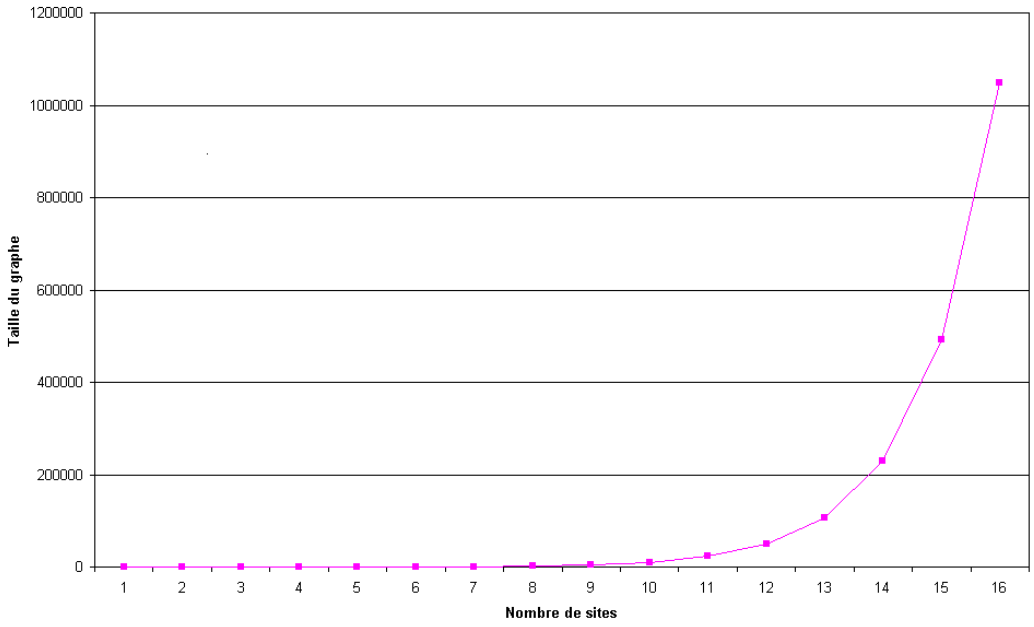


Figure II.3: Scheduler de Milner - Taille du graphe d'accessibilité en fonction du nombre de sites

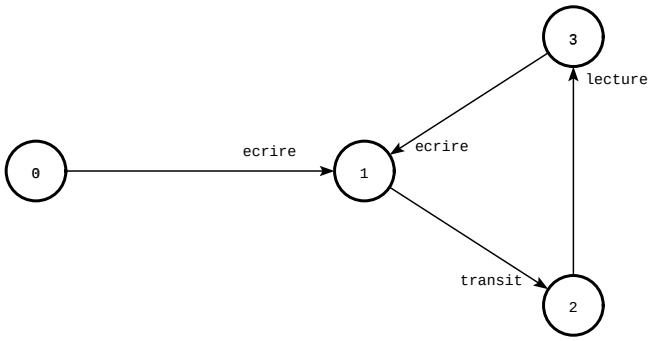


Figure II.4: Écrivain/Lecteur temporisés - Graphe de classes

plus suffisamment d'énergie il ne peut plus faire d'action qui nécessite de l'énergie. Chaque action est caractérisée par un temps minimal (lorsque le planteur est en forme !) et un temps maximal (lorsqu'il n'a pas bien dormi la veille). Nous avons modélisé deux planteurs en parallèle, avec un intervalle $[0, 30]$ pour l'action *dort* pour le premier planteur et un intervalle $[0, 29]$ pour cette même action du second planteur. On observe que le graphe de classes obtenu pour le système composé des deux planteurs a seulement 13 511 classes tandis que le comportement atemporel de ce même système contient 5 089 536 marquages.

Inversement, pour certains systèmes l'ajout de contraintes temporelles fait exploser la taille du graphe de classes. En effet bien que le nombre de comportements soit plus faible que dans le cas atemporel, la description de ces comportements peut être plus complexe.

Pour illustrer cette explosion due aux contraintes temporelles, nous utilisons le réseau de Petri de la figure II.6 : le graphe des marquages comporte 1 seul état, tandis que le comportement temporel met en évidence que la transition t_1 ne peut avoir lieu que toutes les 3000 fois l'action

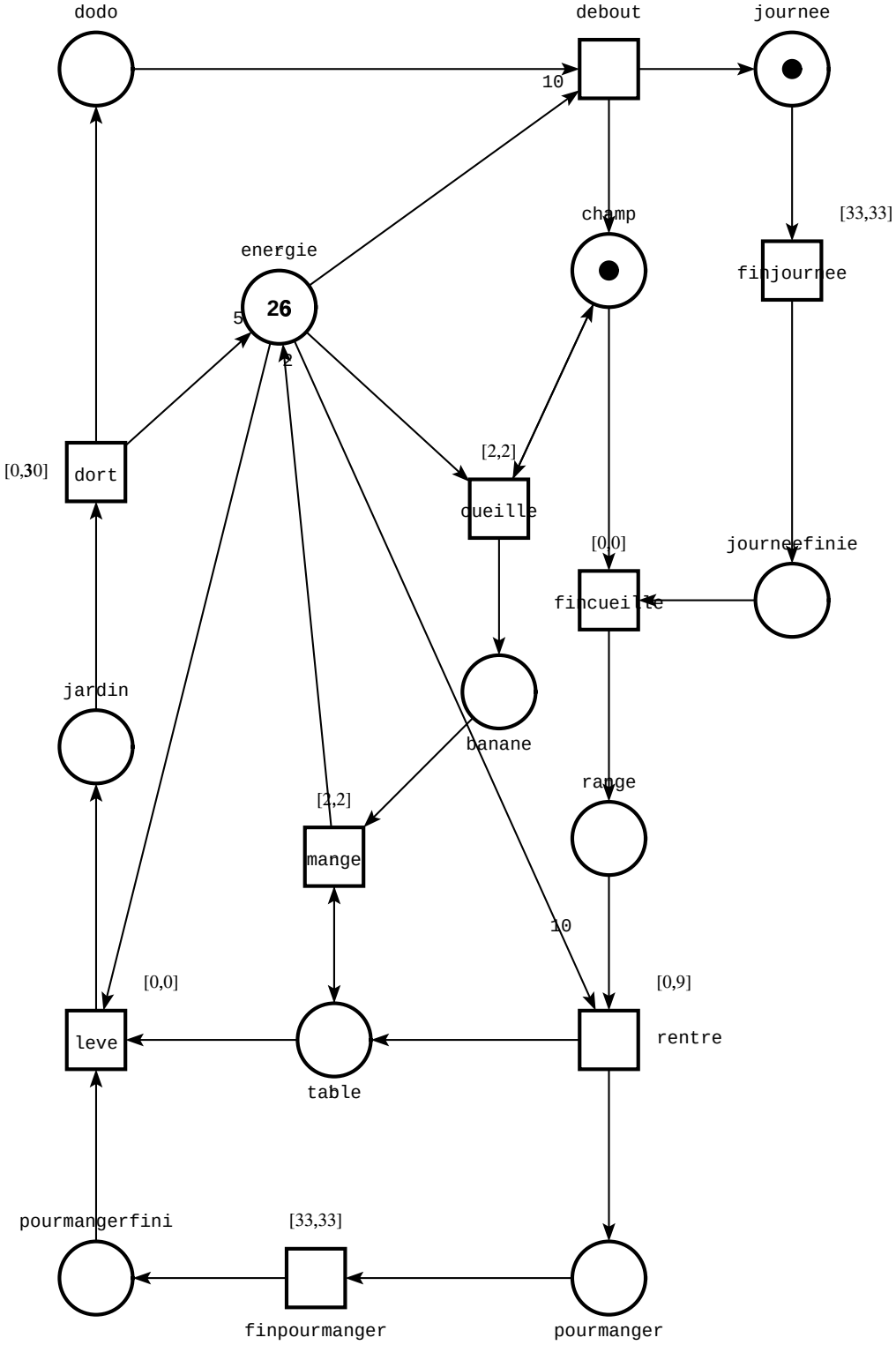


Figure II.5: Les planteurs de bananes avec énergie et temporisations.

Modèle	Nombre de Marquages	Nombre de Classes
Modèle de la figure II.6	1	3002
Les trains de Genrich	2 253	128 736
Planteur de bananes	5 089 536	13 511

Table II.2: Graphe de comportement atemporel et Graphe de classe

t_0 par conséquent le graphe obtenu a 3002 classes.

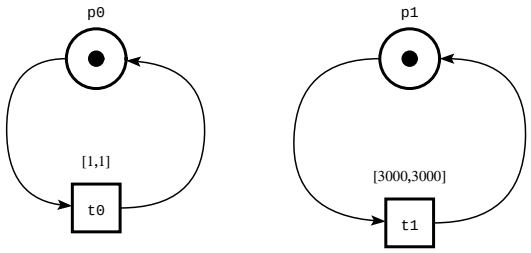


Figure II.6: Exemple d'explosion due à la contrainte temporelle

La Table II.2 récapitule les tailles de quelques graphes obtenus en atemporel et en temporel, pour différents exemples. Les deux premiers exemples illustrent l'explosion du nombre de classes à cause des contraintes temporelles, tandis que le troisième au contraire correspond à un nombre de classes très restreint en limitant fortement le nombre de comportements possibles.

2 Approches pour la maîtrise de l'explosion du nombre d'états

Pour faire face au problème de l'explosion combinatoire, de nombreuses méthodes ont été développées. Le plus souvent celles-ci sont indépendantes, du moins dans leur principe général, des techniques de description et de validation. Il existe deux types de stratégies : la gestion de l'explosion et sa réduction [MV98].

- Pour gérer l'explosion combinatoire, des techniques sophistiquées sont mises en oeuvre pour « contourner » la difficulté de traiter des systèmes de grande taille : compression de graphes et vérification à la volée.
- Pour réduire la complexité du système on tire parti de régularités, de redondances ou de similitudes existant dans la représentation exhaustive.

Il faut noter que ces approches sont souvent complémentaires et peuvent donc être appliquées conjointement pour en cumuler les bénéfices. Nous allons présenter différentes techniques de ces deux types d'approches.

2.1 Gestion de l'explosion

2.1.1 Compression de Graphes

Lorsque les graphes comptent plusieurs millions ou milliards d'états et de transitions, les limites physiques de la mémoire sont vite atteintes. On a recours alors à des techniques de compression de graphes. Les plus efficaces d'entre elles sont basées sur la notion de BDD (Binary Decision

Diagrams) [BCM+90] et permettent dans le meilleur des cas de coder des graphes comptant 10^{20} états. Comme les BDD codent exactement les graphes de comportement, il est théoriquement possible de vérifier toutes les propriétés d'un système sur des représentations à base de BDD. En contrepartie, ils seront infinis si le graphe initial est infini. De plus le taux de compression du graphe dépend fortement de l'ordre des variables. Ainsi, en jouant sur l'ordre des variables, on obtient un BDD de taille soit linéaire soit exponentielle par rapport au nombre de variables. Pour éviter ce problème, des techniques d'ordonnancement de variables ont été mises au point. Cependant, certains graphes ont des BDD exponentiels quel que soit l'ordre des variables (voir la multiplication d'entiers [Bry86]).

2.1.2 Vérification à la volée

Dans le schéma classique, la vérification de systèmes se déroule en deux temps : tout d'abord la construction du graphe de comportement, puis l'analyse de ce dernier. La vérification à la volée consiste à factoriser ces deux tâches : la satisfaction des propriétés est évaluée au cours de l'énumération du comportement.

La vérification à la volée est particulièrement efficace dans le cas où la propriété n'est pas satisfaite par le système. En effet, dès que l'exploration atteint un état d'erreur, elle s'arrête sans visiter le reste du graphe. La vérification à la volée permet donc d'interrompre l'exploration plus rapidement lorsqu'il existe un contre-exemple. Elle permet également de gérer au mieux les ressources mémoires disponibles. En effet elle peut permettre de trouver un contre-exemple du comportement du système, alors que la mémoire n'était pas suffisante pour la construction du comportement complet du système.

Cette technique est utilisable conjointement à toute technique de réduction du graphe, du moment que la technique de vérification utilisée n'a pas besoin d'avoir le graphe complet en une seule fois.

Il peut également être intéressant d'ajouter une heuristique lorsqu'on fait une exploration à la volée. En effet si une heuristique est capable de nous conduire plus rapidement vers le comportement qui ne satisfait pas la propriété, la vérification à la volée sera d'autant plus efficace. L'intérêt d'une telle approche est de ne pas avoir besoin de garantir le guidage, mais de pouvoir implémenter une heuristique qui utilise un calcul local peu coûteux pour essayer de « forcer la chance ».

Cependant lorsque la propriété est vérifiée par le système, la vérification nécessite une exploration complète du graphe : dans ce cas la méthode n'apporte pas de gain ni en temps, ni en mémoire.

Les techniques développées dans la suite de la thèse sont compatibles avec une vérification au vol. Cependant lorsque l'on cherche à comparer deux techniques X et Y, on préfère comparer la taille des graphes construits dans le pire des cas. En effet si on compare les graphes obtenus avec « X+vérification au vol » et « Y+vérification au vol », il sera très difficile de savoir si c'est la méthode qui est plus efficace ou si la vérification au vol a eu « plus de chance » dans un cas que dans l'autre. Par conséquent dans la suite les chiffres donnés dans les tableaux de comparaison sont toujours des tailles de graphes sans utilisation de la technique de vérification au vol.

En conclusion la vérification au vol est **très intéressante d'un point de vue pratique**, puisque elle n'a pas de coût additionnel (pas d'overhead mémoire ou temps) et permet avec un peu de chance d'avoir un résultat beaucoup plus rapidement et peut même permettre d'obtenir un contre-exemple alors que la construction totale aurait échoué par manque de mémoire. Il serait par conséquent très dommage de s'en priver.

2.2 Réduction de l'explosion

Autant les méthodes précédentes prennent le parti ou le risque de construire un graphe exhaustif, autant les techniques de réduction se proposent de ne représenter qu'en partie le comportement du système. Si cette représentation doit être partielle pour éviter l'explosion, elle doit être « suffisamment pertinente » pour en permettre une certaine analyse. Le « compromis » entre le caractère partiel du graphe et la pertinence de l'analyse est trouvé en tirant parti de régularités, de redondances ou de similitudes existant dans la représentation exhaustive.

La figure II.7 présente le principe général des techniques de réduction :

- sur la gauche de la figure est représentée l'exploration classique : partant d'une description formelle du système (exprimée sous forme de réseaux de Petri ou de tout autre formalisme), une exploration exhaustive est conduite, permettant d'obtenir Σ, le graphe de comportement du système. Les propriétés sont vérifiées sur ce graphe (système de transitions étiquetées, structure de Kripke). On note $\mathcal{L}$, le langage d'expression de ces propriétés,
- sur la droite de la figure est représentée l'approche par réduction : partant toujours de la description formelle du système, une exploration partielle conduit à un graphe « réduit » du comportement du système noté Σ_R. Comme précédemment, ce graphe est considéré comme un modèle du programme et permettra la vérification de propriétés exprimées dans un langage d'expression lui-même réduit, $\mathcal{L}_R$. Sans perte de généralité, on peut supposer ici que $\mathcal{L}_R$ est un sous-ensemble de $\mathcal{L}$.

La relation de préservation assure que la vérification de toute formule $\phi \in \mathcal{L}_R$ peut être conduite indifféremment sur le graphe complet ou sur le modèle réduit :

$$\forall \phi \in \mathcal{L}_R \ : \Sigma \models \phi \text{ ssi } \Sigma_R \models \phi$$

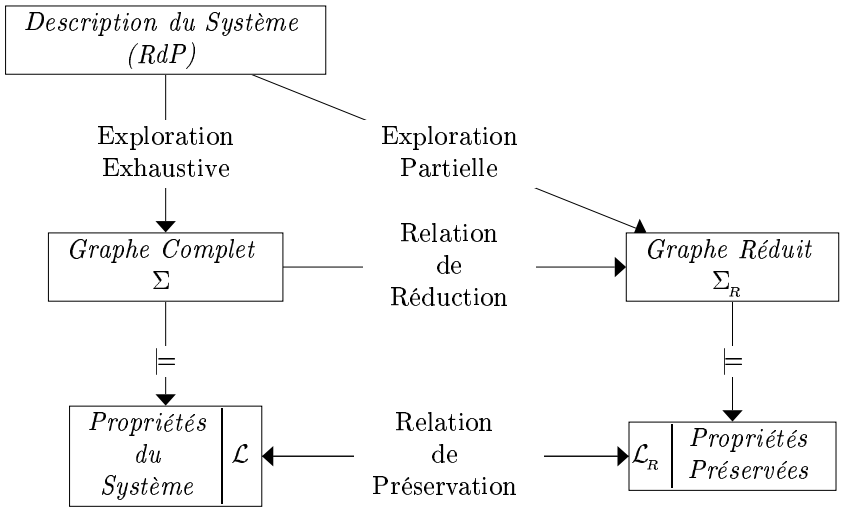


Figure II.7: Principe des techniques de réduction

Les qualités d'une technique de réduction peuvent se résumer par les trois points suivants :

- Fort taux de réduction : le rapport $Size(\Sigma)/Size(\Sigma_R)$ doit être « grand » et précisément d'ordre exponentiel.
- Forte capacité d'analyse : l'ensemble des propriétés vérifiables dans le graphe complet et non vérifiables dans le graphe réduit : $(\mathcal{L} \setminus \mathcal{L}_R)$ doit être le plus petit possible.

– « Simple » à réaliser : Deux critères doivent être pris en compte : le temps et l'espace mémoire nécessaires à la construction du graphe réduit par rapport à ceux nécessaires à la construction du graphe complet.

Il va de soi que les deux premiers points sont orthogonaux ; dans la pratique il s'agit de trouver le « juste compromis » entre le taux de réduction et la capacité d'analyse. Une façon d'approcher ce compromis consiste à « contrôler » la réduction par la propriété que l'on cherche à vérifier.

-Méthode d'abstraction
-Méthode symétrique
-Méthode ordre partiel
* Technique par choix
ensembles persistants
ensembles dormants
ensembles amples
* Technique par pas
graphe de pas couvrants
analyse d'accessibilité simultanée

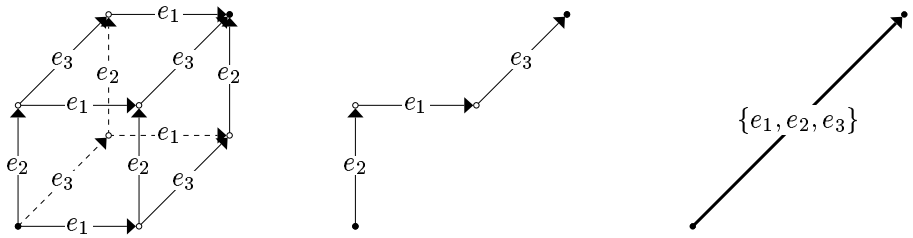
Table II.3: Maîtrise de l'explosion combinatoire par réduction

Le tableau II.3 récapitule différentes approches pour la maîtrise de l'explosion combinatoire par réduction. Les méthodes d'abstraction permettent de limiter la prolifération d'états différant les uns des autres par des détails éventuellement insignifiants au vu des propriétés à vérifier. En particulier lors d'une énumération exhaustive, les variables du système prennent toutes les valeurs possibles de leur domaine. Les méthodes d'abstraction éliminent ces détails dans la phase de validation.

Les méthodes symétriques réduisent les redondances causées par les symétries du système. En effet les systèmes répartis contiennent souvent un grand nombre de composants symétriques (possédant des comportements similaires et situés à des endroits équivalents dans l'architecture).

Notre travail s'intéresse plus particulièrement aux différentes techniques dites « ordres partiels ». Les sémantiques classiques représentent le parallélisme par l'entrelacement d'actions, ainsi deux actions parallèles notées $A \mid B$ seront représentées par le « losange » $A.B + B.A$: dans ces sémantiques l'indépendance causale de A et de B est approximée par le choix entre deux contraintes causales A puis B ou B puis A. Cette approximation, en multipliant les entrelacements d'actions contribue à l'explosion combinatoire. Lorsque deux (n) composants offrent une action en parallèle, la sémantique d'entrelacement représente ce comportement par un losange (un hypercube dans le cas général) où les différents chemins confluent vers un même état. La figure II.8 illustre le graphe construit pour trois transitions indépendantes e_1,e_2,e_3. Comme tous ces chemins confluent vers le même état, l'idée directrice commune à ces approches consiste à n'explorer qu'un chemin particulier parmi tous les chemins équivalents possibles. Cette idée est illustrée dans le second schéma de la figure II.8. Intuitivement, les états intermédiaires du chemin ne sont pas significatifs puisque un autre chemin aurait pu être choisi ; aussi la méthode des graphes de pas couvrants exploite cette propriété pour ne construire qu'un seul pas allant de l'état initial à l'état d'arrivée.

Les sections suivantes vont suivre ce déroulement en présentant en premier lieu les notions classiques de relation d'indépendance dans les systèmes de transitions [WG93] et montrer un exemple d'approximation structurelle de cette relation dans le cas des réseaux de Petri. Nous rappelons ensuite la notion d'alphabet concurrent et de traces de Mazurkiewicz, qui sont les fondements des approches « ordre partiel ». Les principes généraux des techniques fondées sur



	Exhaustive	1 chemin / Trace	**Pas**
États	2^n	$(n+1)$	2
Transitions	$n \times 2^{n-1}$	n	1

Figure II.8: Graphe d'états obtenu pour 3 transitions indépendantes

cette approche sont finalement exposés.

3 Approches « ordre partiel »

3.1 Fondements des approches « ordres partiels »

Définition 34 *Relation d'indépendance [WG93]*

Soit un STE $\Sigma = \langle S, s_o, T, \rightarrow \rangle$ *et* $\wr$ *une relation binaire sur l'ensemble des transitions de* Σ *(*$\wr \subset T \times T$*) :* $\wr$ *est une relation d'***indépendance** *pour* Σ *ssi*

$\forall s, s_1, s_2 \in S, \forall t_1, t_2 \in T \ : [t_1 \neq t_2, s \xrightarrow{t_1} s_1, s \xrightarrow{t_2} s_2 \text{ et } t_1 \wr t_2] \Rightarrow \exists s' \in S \ : s_1 \xrightarrow{t_2} s' \text{ et } s_2 \xrightarrow{t_1} s'$

Si deux transitions indépendantes t_1 et t_2 sont possibles en s ($s \xrightarrow{t_1} s_1$ et $s \xrightarrow{t_2} s_2$) alors t_2 est possible en s_1 et t_1 est possible en s_2, de plus il existe un unique état s' vérifiant à la fois $s_1 \xrightarrow{t_2} s'$ et $s_2 \xrightarrow{t_1} s'$

Définition 35 *Relation de conflit et fermeture transitive*

La relation complémentaire de $\wr$ *est appelée relation de conflit ou de dépendance. On la note* $\#$.

La fermeture réflexive et transitive de la relation de conflit est notée $[\#]$. *Cette relation est dite de* conflit faible.

$[\#]$ est une relation d'équivalence et $\# \subset [\#]$. Sa relation complémentaire $[\#]^C$, est dite d'*indépendance forte* et vérifie $[\#]^C \subset \wr$

La notion d'indépendance que nous venons de voir est définie directement sur le STE. Celle-ci n'est donc pas directement utilisable puisque nous ne voulons pas construire le STE. On peut par contre obtenir des approximations de cette relation en travaillant directement sur la description du système et sur la sémantique transitionnelle du formalisme dans lequel elle est écrite.

Nous donnons ici un exemple d'approximation de cette relation d'indépendance dans le cas des réseaux de Petri [Rei85].

Définition 36 *Approximation de la relation d'indépendance*

Soit $R = \langle P, T, Pre, Post \rangle$ *un réseau Place/Transition,* m_0 *un marquage initial et* $STE(R, m_0)$ *le graphe des marquages accessibles associé au réseau marqué* $\langle R, m_0 \rangle$.

La relation $| \subset T \times T$ *définie par :* $t_1 \mid t_2$ *ssi* ${}^\bullet t_1 \cap {}^\bullet t_2 = \emptyset$ *est une relation d'indépendance pour* $STE(R, m_0)$.

Cette relation d'indépendance est une relation d'indépendance statique, seule la structure du réseau de Petri est prise en compte en considérant les pré-conditions des transitions. Cette relation est donc indépendante de l'état courant et n'est calculée qu'une seule fois au début de l'exploration. La relation $\#\!\!\#$ est la fermeture transitive de la relation de conflit $\#$, qui est la complémentaire de cette relation d'indépendance.

Définition 37 *Ensemble des transitions en conflit avec une transition t :*

$$\#\!\!\#(t) = \{t' \in T \ : t \#\!\!\# t'\}$$

Définition 38 *Transition libre*

Une transition est libre si elle n'est en conflit avec aucune autre transition qu'elle même.

$$t \text{ libre ssi } \#\!\!\#(t) = \{t\}$$

Les transitions libres sont particulièrement intéressantes puisque elles ne peuvent pas être désensibilisées par le tir d'une autre transition.

Dans un contexte différent, Mazurkiewicz [Maz86] a défini une relation d'équivalence - dont les classes sont appelées « traces » - opérant sur les séquences d'actions : deux séquences d'actions sont équivalentes si et seulement si elles peuvent être obtenues l'une à partir de l'autre par une série de permutations d'actions adjacentes et indépendantes.

Définition 39 *Alphabet concurrent*

Un alphabet concurrent est un couple $\mathcal{E} = (\alpha, \#)$ où α est un alphabet et $\#$ une relation de dépendance *dans α ($\#$ est réflexive et symétrique). La relation complémentaire $\#^C$ (i.e. $\alpha^2 \setminus \#$) est appelée relation d'indépendance dans α.*

Définition 40 *Traces de Mazurkiewicz*

Pour un alphabet concurrent $\mathcal{E} = (\alpha, \#)$, on considère $\approx_\mathcal{E}$ la relation définie sur $\alpha^* \times \alpha^*$ par $\forall U, V \in \alpha^*, \forall a, b \in \alpha, U.a.b.V \approx_\mathcal{E} U.b.a.V$ ssi $(a, b) \in \#^C$. Par construction, $\approx_\mathcal{E}$ est réflexive et symétrique.

L'équivalence de traces de Mazurkiewicz, notée $\equiv_\mathcal{E}$, peut être définie comme la fermeture transitive de la relation $\approx_\mathcal{E}$. Ses classes d'équivalence sont appelées des traces sur $\mathcal{E}$. $[w]_\mathcal{E}$ dénote la trace sur $\mathcal{E}$ générée par le mot w

Propriété 3 *Propriétés des traces*

$\equiv_\mathcal{E}$ est la plus petite congruence dans le monoïde $(\alpha, ., \epsilon)$[1] vérifiant : $(a, b) \in \#^C \Rightarrow a.b \equiv_\mathcal{E} b.a$

Propriété 4 *Fondement des approches ordres partiels*

Pour un STE Σ, $\wr$ une relation d'indépendance sur Σ et l'alphabet concurrent $(T, \wr^C)$:

$$\forall s \in S, w_1, w_2 \in T^* \ : s \stackrel{w_1}{\Rightarrow} s_1 \text{ et } s \stackrel{w_2}{\Rightarrow} s_2 \text{ et } [w_1]_\mathcal{E} = [w_2]_\mathcal{E} \Rightarrow s_1 = s_2$$

La propriété précédente fonde les approches d'ordres partiels en reliant la notion d'indépendance dans les STE et la notion de trace de Mazurkiewicz : les chemins possédant la même trace conduisent aux mêmes états. En conséquence, on peut « sous certaines conditions » réduire l'exploration en ne prenant en compte qu'un seul chemin par trace. On limite ainsi l'une des causes de l'explosion combinatoire : la représentation de la concurrence par l'entrelacement d'actions.

[1] L'opérateur . désigne l'opération interne dans α et ϵ l'élément neutre pour . dans α

Considérons par exemple le réseau de Petri de la figure II.9. Au milieu de la figure est représenté son graphe d'accessibilité. Le parallélisme des transitions $t1$, $t2$ et $t3$ est représenté par toutes les séquences possibles formées sur les permutations de ces trois transitions. Toutes ces séquences conduisent au même état 4. Par conséquent une seule de toutes ces séquences serait vraiment utile ; par exemple celle choisie sur la figure de droite.

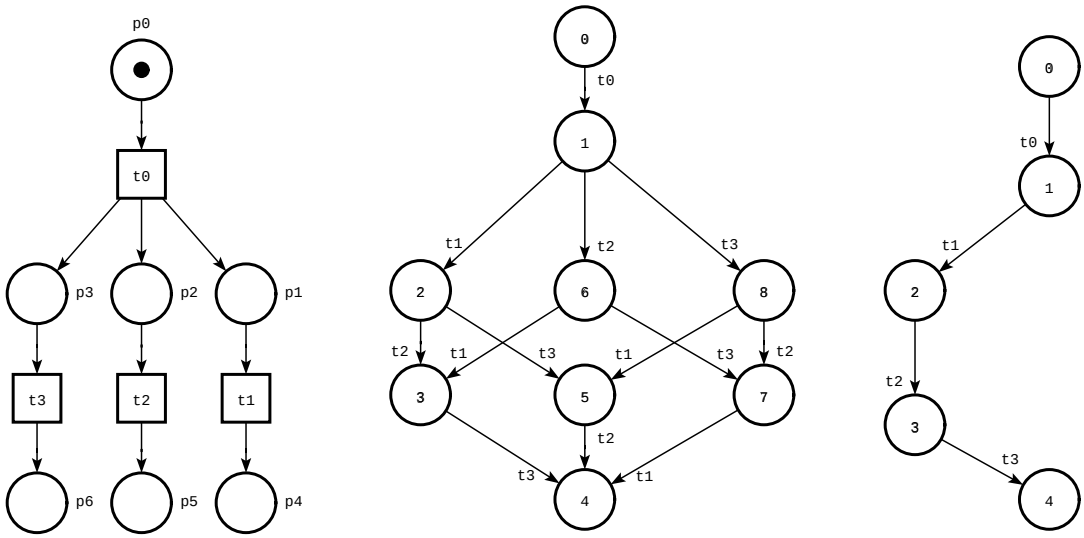


Figure II.9: Exemple de traces équivalentes

Cependant cette réduction ne peut avoir lieu que « sous certaines conditions », que nous précisons au cours des prochains chapitres, suivant l'approche utilisée et la propriété que l'on souhaite vérifier. Une situation souvent rencontrée est appelée « situation de confusion » (ou de conflits différés) [Rei85] et illustre la difficulté de ce genre de réductions : autant les chemins possédant la même trace confluent vers le même état, autant rien ne permet d'assurer que les états intermédiaires constituant ces chemins soient eux-mêmes « équivalents ».

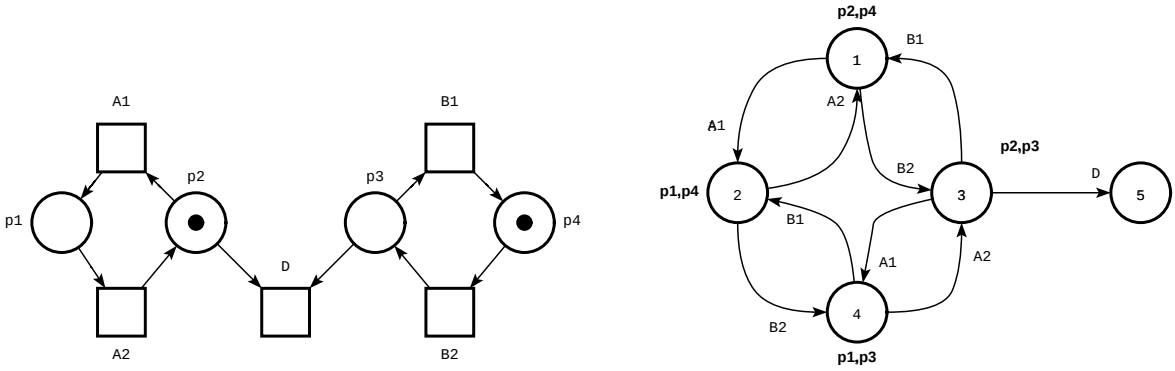


Figure II.10: Situation de confusion - Réseau de Petri et graphe d'accessibilité

Considérons le réseau de Petri de la figure II.10 et son graphe d'accessibilité (STE) donné à droite. Les transitions A_1 et B_2 sont indépendantes ($^\bullet A_1 \cap {}^\bullet B_2 = \emptyset$). Partant du marquage $\{p_2, p_4\}$ les séquences d'action indépendantes $A_1.B_2$ et $B_2.A_1$ confluent toutes deux vers le mar-

quage $\{p_1, p_3\}$. Pour autant les états intermédiaires qui constituent cette séquence ne sont pas équivalents du point de vue de la possibilité de blocage. En effet, en suivant la séquence $B_2.A_1$ on passe par le marquage $\{p_2, p_3\}$ qui offre la possibilité de tirer la transition D (conduisant le système dans un état de blocage). Si on suit la séquence $A_1.B_2$ il n'est pas possible de franchir la transition D et d'atteindre l'état de blocage. Le problème vient de l'action D qui est en conflit avec A_1 et qui n'est pas exécutable mais seulement atteignable depuis $\{p_2, p_4\}$. D ne devient exécutable qu'après avoir exécuté B_2.

Cet exemple du cas de confusion montre que suivant la propriété à vérifier, l'indépendance entre les transitions n'est pas suffisante pour pouvoir appliquer les réductions. Dans la suite en fonction de l'approche et de la propriété souhaitée, nous préciserons les conditions d'application de chaque réduction pour tenir compte du cas de confusion.

3.2 Préservation des blocages

Plusieurs techniques permettent de construire un graphe réduit préservant les états de blocage. Nous avons regroupé ces techniques en deux catégories : Les techniques par « choix » qui dans un état donné ne vont pas explorer tous les successeurs, mais vont choisir un sous-ensemble de transitions à explorer. Les techniques par « pas » vont explorer toutes les transitions mais en les regroupant en « pas ».

3.2.1 Techniques par « choix »

Ensembles persistants La première technique de construction d'un graphe réduit que nous allons détailler est la méthode des ensembles persistants. Cette technique permet de construire un graphe réduit qui conserve les états de blocage du système.

Les ensembles persistants sont un cas particulier des ensembles stubborns [Val88a] dans lesquels toutes les transitions sont sensibilisées. L'exploration standard par ensembles persistants conserve les blocages ; de nombreuses extensions ont été proposées dans le but de préserver d'autres classes de propriétés [Val90, Val93] [GW93], nous en verrons dans la suite avec les ensembles proviso et les ensembles amples.

Définition 41 *Ensemble persistant [WG93]*

Un ensemble $E_P \subseteq T$ de transitions est persistant dans un état s si et seulement si toutes les transitions qui ne sont pas dans E_P et sensibilisées dans s ou dans un état accessible depuis s en tirant des transitions qui ne sont pas dans E_P, sont indépendantes de toutes les transitions de E_P, i.e. ssi :

$$\forall t \in E_P \ : s \xrightarrow{t} \ et \ \forall w \in \overline{E_P}^* \ : s \stackrel{w}{\Rightarrow} s' \Rightarrow w \wr E_P$$

Avec : $\overline{E_P} = T \backslash E_P$ et $w \wr E_P$ ssi $\forall t_1 \in w$ et $\forall t_2 \in E_P, t_1 \wr t_2$

A partir de l'algorithme d'exploration classique présenté dans la table I.1, nous allons remplacer l'appel de la fonction $develop\_exhaustive(Enabled(s))$, par l'appel à une exploration partielle par ensemble persistant $develop\_GP(Enabled(s))$. Cette fonction générique est définie dans la table II.4. Cet algorithme sera noté GP (Graphe Persistant) dans la suite. Il est similaire à un algorithme standard d'exploration exhaustive, mis à part le fait que l'ensemble des transitions tirées depuis un état s est déterminé par une fonction E_P qui choisit un ensemble persistant dans l'état s : $E_P(s) \subseteq Enabled(s)$.

Proposition 1 *L'exploration par GP conserve les blocages [Val88a]*

develop_GP(E) :
$E_P \leftarrow E_P(E)$
$develop\_exhaustive(E_P)$

Table II.4: Algorithme de construction d'un graphe persistant (GP)

Démonstration La démonstration est donnée dans [Val88a]. Un ensemble persistant contient au minimum toutes les transitions qui doivent être explorées dans un état spécifique afin de découvrir tous les blocages éventuels. Si E_P est un ensemble persistant dans l'état s, alors aucune transition de E_P ne peut être désensibilisée par une séquence de tirs de transitions qui ne sont pas dans E_P.

La propriété suivante tirée de [God90] permet de déduire aisément un algorithme de construction d'ensembles persistants.

Propriété 5 *Si* $\#(t) \subseteq Enabled(s)$ *Alors* $\#(t)$ *est persistant dans* s.

Démonstration Par construction de $\#(t)$ toutes les transitions qui ne sont pas dans l'ensemble ne sont en conflit avec aucune transition de l'ensemble.

Dans un état donné, pour chaque transition t sensibilisée, on peut construire l'ensemble $\#(t)$. La propriété précédente permet de déduire si l'ensemble ainsi construit est persistant ou pas. On choisit alors un ensemble persistant, parmi tous les ensembles persistants ainsi construits (les critères utilisés seront détaillés ci-après).

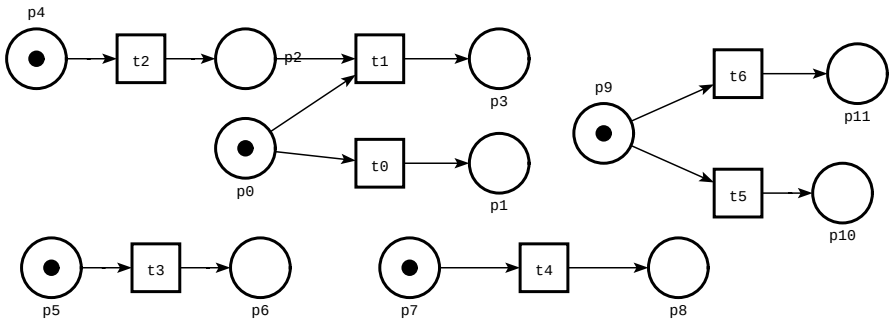


Figure II.11: Réseau de Petri d'étude

Nous allons prendre le réseau de Petri de la figure II.11 comme exemple. Le graphe de comportement exhaustif de ce réseau de Petri comporte 60 marquages et 160 transitions avec 4 états de blocage.

D'après la structure du réseau de Petri et la relation d'indépendance définie par des préconditions disjointes ($^\bullet t_i \cap {}^\bullet t_j = \emptyset$), on a $\#(t0) = \#(t1) = \{t0, t1\}$, $\#(t2) = \{t2\}$, $\#(t3) = \{t3\}$, $\#(t4) = \{t4\}$, $\#(t5) = \#(t6) = \{t5, t6\}$. Dans l'état initial s_0, $Enabled(s_0) = \{t0, t2, t3, t4, t5, t6\}$, donc les ensembles précédents inclus dans l'ensemble $Enabled(s_0)$ sont des ensembles persistants : les ensembles $\{t2\}, \{t3\}, \{t4\}$ et $\{t6; t5\}$ sont persistants. D'autres ensembles persistants peuvent être trouvés, il est très simple de montrer par exemple que l'union d'ensembles persistants est persistante. L'ensemble des transitions sensibilisées dans un état s (noté $E(s)$) est lui aussi un ensemble persistant (qui n'apporte aucune réduction puisque c'est l'ensemble utilisé pour construire le graphe complet) on dit alors que l'état s est complètement exploré (« fully expanded »).

Les ensembles persistants sont calculés et choisis en fonction d'informations locales à l'état, il est donc très difficile de connaître l'influence réelle sur la taille globale du graphe. Il est par conséquent impossible de savoir localement quel est le « meilleur »[2] ensemble persistant. Aussi différentes heuristiques ont été proposées [God90, Ove81, Val89] pour définir la fonction E_P permettant de choisir un ensemble persistant. Une stratégie souvent utilisée est de prendre le plus petit ensemble persistant que l'on sache calculer. Cette stratégie minimise le branchement dans chaque état, ce qui permet de minimiser localement le nombre de successeurs construits. Cette instance du GP sera dénommée l'algorithme GP_{min} dans la suite. La technique des ensembles persistants a été implémentée dans l'outil *Tina* à l'aide de l'algorithme de la table II.4 et de l'instance GP_{min} pour la fonction permettant de calculer un ensemble persistant. Cette implémentation a été intégrée à la distribution de *Tina*.

L'utilisation de GP_{min} sur notre exemple de la Figure II.11 produit le graphe de la Figure II.12. Le dessin du graphe en gras est dessiné sur le graphe complet en gris clair. On peut en particulier noter les 4 états de blocages vers lesquels toutes les exécutions convergent obligatoirement.

Cependant, minimiser le branchement est une optimisation locale ; dans certains cas le choix d'un ensemble persistant plus grand conduit à un graphe final plus petit [Val88b]. De plus il est important de remarquer que l'ensemble des transitions sensibilisées admet généralement plusieurs ensembles persistants distincts et donc éventuellement de cardinalités différentes. Par conséquent, pour calculer le plus petit ensemble persistant, il est nécessaire d'examiner toutes les transitions sensibilisées comme candidates pour la génération de l'ensemble persistant, ce qui peut entraîner un surcoût de calcul.

De plus, si différents ensembles persistants sont minimaux, le choix de l'un d'entre eux devient arbitraire. Par exemple dans le cas du réseau de Petri de la Figure II.11, dans l'état initial, les trois ensembles persistants $\{t_2\}, \{t_3\}, \{t_4\}$ ont le même cardinal. Le choix de l'ensemble $\{t_3\}$ dans le graphe de la Figure II.12 est complètement arbitraire : avec le même modèle et le même programme on peut construire des graphes différents si on l'exécute plusieurs fois. Cela peut non seulement être très troublant pour l'utilisateur, mais complique également les comparaisons des tailles des graphes construits par différentes techniques.

Ensembles dormants (Sleep Set) Comme le parallélisme est représenté par l'entrelacement d'actions, on retrouve très souvent des états déjà explorés par d'autres chemins comme illustré dans la figure II.8. Pour la recherche d'états de blocages, il est inutile de retourner dans des états déjà rencontrés. La technique des ensembles dormants consiste à se rappeler des transitions déjà explorées (dans un autre chemin) pour ne pas les explorer à nouveau. On dit qu'elles sont « endormies ».

Cette technique n'exploite donc pas une information statique de la structure du système, mais exploite le passé de l'exploration pour améliorer l'exploration actuelle. Pour cela à chaque état est associé un ensemble de transitions dormantes. Lorsqu'on tire une transition t depuis un état s, la transition t est ajoutée à l'ensemble dormant de l'état s. De plus le tir de la transition t ayant permis d'atteindre un état s', deux cas se présentent : dans le cas où l'état atteint s' est un nouvel état, l'ensemble dormant qu'on lui associe est celui de s auquel on enlève toutes les transitions en conflit avec t (soit l'ensemble $\#(t)$) ; dans le cas où l'état existait déjà alors son ensemble dormant prend l'intersection de l'ensemble dormant qu'il possédait déjà et de l'ensemble dormant défini précédemment pour les nouveaux états.

Cette technique ne permet pas de réduire le nombre d'états construits mais uniquement de ne pas tirer certaines transitions. Nous ne pouvons donc pas montrer cette construction sur

[2] Dans le sens, où notre objectif est de construire un graphe le plus petit possible

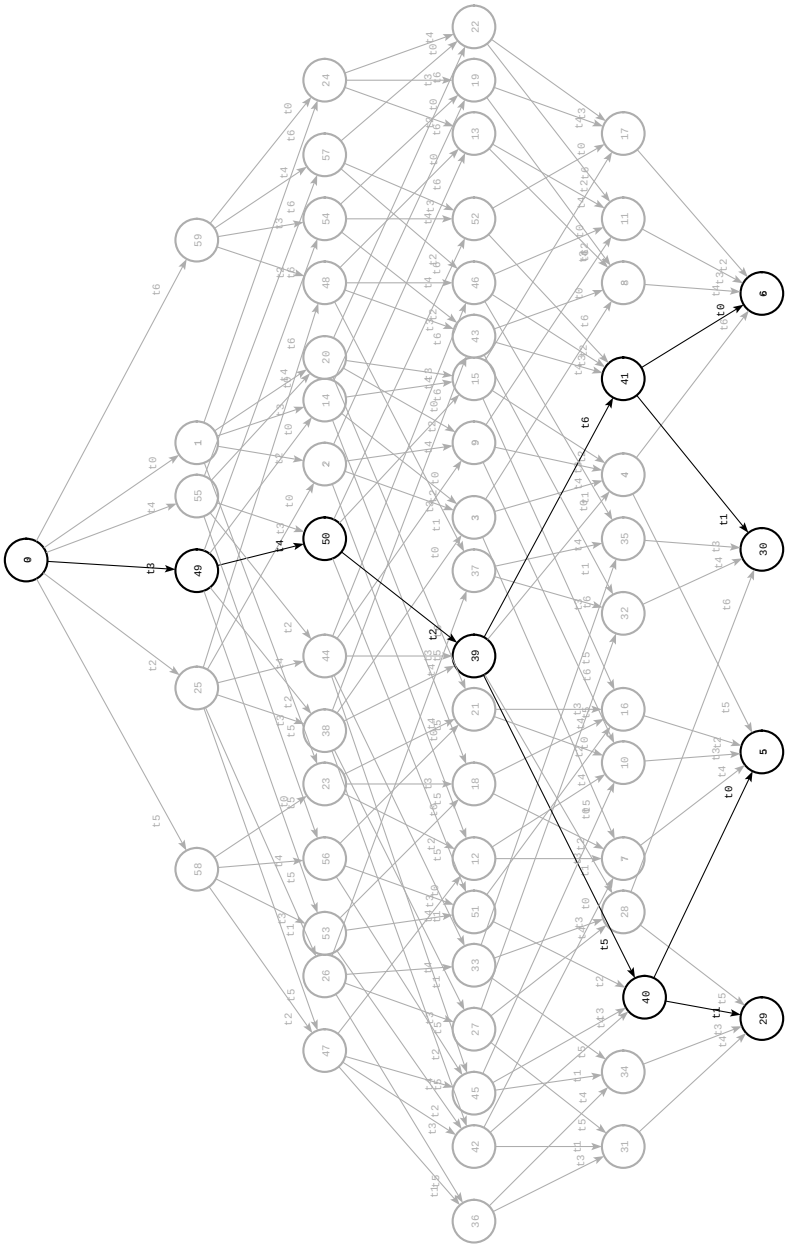


Figure II.12: GP_{min} : GP calculé avec une stratégie minimisant le branchement

l'exemple de la figure II.11, puisque son graphe exhaustif comporte 60 états. On le montre donc sur un exemple beaucoup plus petit avec deux transitions en parallèles (Figure II.13). Dans l'état $s0$ on commence par explorer la transition t_0 donc l'ensemble dormant de l'état $s0$ devient $\{t_0\}$; lorsqu'on tirera la transition t_1 depuis l'état $s0$, on atteindra l'état $s2$ qui prendra alors l'ensemble dormant $\{t0\}$. Donc depuis cet état la transition t_0 ne sera pas explorée.

La technique des ensembles dormants ne réduit pas le nombre d'états du graphe construit, mais uniquement le nombre de transitions. Diminuer le nombre de transitions construites a un gain indirect lors de la construction : lors de l'exploration décrite table I.1, pour chaque transition explorée, on construit l'état atteint et on cherche si cet état existait déjà dans la partie construite. Le fait d'explorer moins de transitions permet donc de diminuer le nombre d'états calculés et

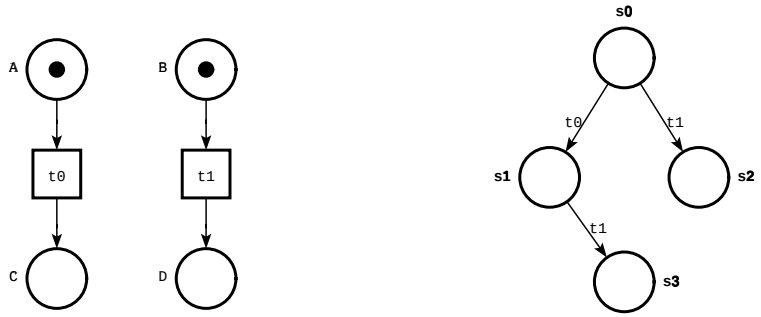


Figure II.13: Exemple avec les ensembles dormants.

recherchés. Le coût à payer pour ce gain, correspond à la gestion des ensembles dormants.

L'approche des ensembles dormants peut-être cumulée avec l'approche des ensembles persistants, cependant comme les deux approches exploitent la même techniques de réduction, elles arrivent généralement à réduire aux mêmes endroits : les ensembles persistants ne vont pas aller construire les parties où les ensembles dormants auraient évité la construction de transitions.

3.2.2 Techniques par « Pas »

Dans les techniques par pas, toutes les transitions sensibilisées sont tirées, mais les évènements indépendants sont (autant que possible) regroupés pour constituer un seul pas de transitions, dont le tir est atomique. L'idée de regrouper le tir de transitions indépendantes n'est pas récente. On trouve dans [Rei85] une notion de graphe de pas où l'on ajoute aux transitions du STE classique tous les pas de transitions possibles. Cette approche est aussi utilisée pour favoriser la convergence dans certaines méthodes d'accélération [BF00]. La principale différence est liée au fait que, pour réduire la taille du graphe, les pas de transitions ne sont pas ajoutés au STE classique mais substitués aux transitions.

Graphes de pas couvrants (GPC) Nous présentons ici rapidement les graphes de pas couvrants, introduits dans [VAM96]. Une présentation détaillée et illustrée est donnée dans le chapitre III.

La décision locale de constituer un pas d'exploration n'a de sens que dans la mesure où l'on sait que toutes les linéarisations de ce pas auraient conduit à des états intermédiaires dont l'exploration n'est pas utile vis à vis de la classe de propriétés que l'on considère.

Un « pas de transitions » est un ensemble de transitions indépendantes entre elles. L'ensemble des pas possibles sur l'ensemble des transitions T par rapport à la relation d'indépendance $\wr$ est noté $Step(T, \wr)$. La relation d'accessibilité $\rightarrow$ est étendue aux pas de transitions par :

$$s \overset{\emptyset}{\twoheadrightarrow} s \text{ et } \forall s, s' \in S, F \in Step(T, \wr) \ : s \overset{F}{\twoheadrightarrow} s' \text{ ssi } \forall t \in F \ : s \xrightarrow{t} s_1 \wedge s_1 \overset{F\setminus\{t\}}{\twoheadrightarrow} s'$$

La relation d'indépendance $\wr$ est également étendue aux pas par

$$F \ \wr \ w \text{ ssi } F \times ||w|| \subseteq \wr$$

avec $||w||$ l'ensemble des transitions apparaissant dans la séquence w.

Définition 42 *Graphe de Pas Couvrant*
Soit $\Sigma = \langle S, s_o, T, \rightarrow \rangle$ *un STE,* $\wr$ *une relation d'indépendance sur* Σ *et* $\# = \wr^C$.
$\Sigma_R = \langle S_R, s_o, T_R, \rightarrow_R \rangle$ *est un* **Graphe de Pas Couvrant** *pour* Σ*, par rapport à* $\wr$*, ssi*
(1) $S_R \subseteq S$
(2) $T_R \subseteq Step(T, \wr)$
(3) $\forall s, s' \in S \cap S_R, \forall E \in T_R \ : s \xrightarrow{E}_R s'$ *implique* $s \overset{E}{\twoheadrightarrow} s'$
(4) $\forall s \in S \cap S_R, \forall s' \in S$:

$$s \xrightarrow{w} s' \ \ implique \ \ \begin{cases} \exists s" \in S_R, \exists w' \in T^*, \exists P_{w.w'} \in {T_R}^* \ : \\ s' \xrightarrow{w'} s", s \xrightarrow{P_{w.w'}}_R s" \ \ et \ [w.w']_{(T,\#)} = [P_{w.w'}]_{(T,\#)} \end{cases}$$

La condition (1) signifie que chaque état du graphe de pas est un état du STE standard. La condition (3) que chaque pas du GPC correspond à une séquence de tir dans le STE standard. Finalement, (4) exprime une « condition de couverture » entre les séquences de tir du STE standard et les séquences de pas du GPC : toute séquence du STE peut être étendue de telle sorte qu'elle soit couverte par une séquence de pas dans le GPC. Remarquons que tout STE peut être vu comme un GPC en prenant $\wr = \emptyset$.

Pour la mise en oeuvre, nous utilisons à nouveau l'algorithme classique dans lequel nous remplaçons l'exploration exhaustive par l'appel de la fonction définie dans la Table II.5. Cet algorithme permet le calcul d'un GPC. L'algorithme est paramétré par une relation d'indépendance $\wr$. Les transitions sensibilisées sont séparées en deux sous-ensembles grâce aux fonctions T_U et T_M. T_u contient les transitions à explorer d'une façon standard et T_m contient celles dont l'exploration sera menée par des pas. Ces transitions seront nommées « fusionnables » dans la suite. Π_{T_M} est l'ensemble des pas de transitions, construit par la fonction Π.

```
develop_GPC(E):
        Tu ← TU(E, ≀)
        Tm ← TM(E, ≀)
        develop_exhaustive(Tu)
        Π_Tm ← Π(Tm, ⧣)
        develop_by_step(Π_Tm)
```

```
develop_by_step(Π):
    for each π in Π do
        s' ← step_fire(s, π)
        G ← G ∪ {< s, π, s' >}
        if s' ∉ S then
            S ← S ∪ {s'}
            Ajouter(Tmp, s')
```

Table II.5: Algorithme de construction d'un Graphe de Pas Couvrants (GPC)

La fonction *develop_by_step*(Π) est utilisée pour explorer les pas construits dans l'état courant les uns après les autres.

Pour la mise en oeuvre de cet algorithme dans l'outil *Tina* nous avons utilisé l'instanciation donnée dans [VAM96] en utilisant l'orthoproduit[3] pour définir les pas de conflits croisés donnés dans la table II.6 avec $E/⧣$ l'ensemble quotient de E par la relation ⧣.

La construction du graphe de pas de conflits croisés sur notre exemple de la Figure II.11 produit le GPC de la Figure II.14 : les pas sont représentés par la suite des transitions le constituant au dessus de la flèche. Dans l'état initial les transitions t_2, t_3, t_4 sont indépendantes donc on peut construire un pas les contenant, par contre t_5 et t_6 sont en conflit, donc deux pas différents sont construits pour chacune d'elles. La transition t_0 est en conflit avec la transition t_1 qui n'est pas

[3]L'orthoproduit [PF90] de $\mathbb{E} = \{E_1, E_2, \ldots E_n\}$ est l'ensemble $\Pi_C(\mathbb{E}) = \{\{e_1, e_2, \ldots e_n\} \mid (e_1, e_2, \ldots e_n) \in E_1 \times E_2 \ldots \times E_n\}$.

$T_M(E,\wr) = \{t \in E \mid t'[\#]t \Rightarrow t' \in E\}$
$T_U(E,\wr) = E \setminus T_M(E,\wr)$
$\Pi(E,\#) = \Pi_C(E/\#)$

Table II.6: Définitions des fonctions pour l'utilisation des conflits croisés

sensibilisée, c'est typiquement la situation du cas de confusion, par conséquent la transition t_0 a été mise dans l'ensemble T_U et aucun pas n'est construit avec elle.

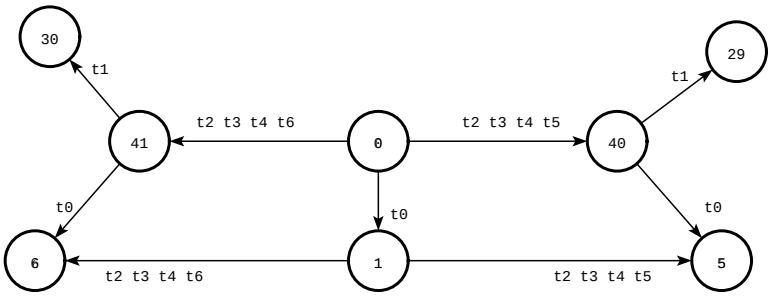


Figure II.14: GPC utilisant les conflits croisés

Simultaneous Reachability analysis (SRA)

L'approche SRA consiste aussi à construire des pas de transitions pour limiter l'explosion combinatoire. Une première version de SRA a été proposée par Rubin et West [Wes86] mais uniquement pour deux processus communiquant, en permettant la détection de tous les blocages et de toutes les réceptions non spécifiées[4]. Itoh et Ichikawa [II83] ont proposé une technique d'analyse de graphe d'accessibilité réduit dans laquelle les états sont générés en exécutant simultanément une transition de chaque processus afin de détecter les blocages et un sous-ensemble particulier de réceptions non spécifiées. La méthode est cependant restreinte aux protocoles dans lesquels les processus ne contiennent que des boucles contenant l'état initial.

Les approches SRA et GPC utilisent une méthode analogue pour réduire la taille du graphe : grouper le tir de transitions. Par conséquent les difficultés se retrouvent à des endroits similaires (par exemple la détection du cas de confusion). Les moyens mis en oeuvre pour résoudre ces difficultés sont différentes et n'exploitent pas les mêmes directions. Par exemple dans le cas du réseau de Petri de la figure II.15, qui présente le cas typique du cas de confusion, tandis que l'approche GPC se résigne à ne pas faire de pas, l'approche SRA préfère faire une branche d'exploration classique et un pas. La figure II.16 illustre le graphe construit par la technique SRA pour le réseau de Petri de la figure II.15.

Pour comparaison avec la technique des graphes de pas couvrants, on ajoute à l'exemple précédent une transition t_3 indépendante. La figure II.17 donne le graphe exhaustif, tandis que la figure II.18 donne le graphe construit par la technique SRA à gauche et par la technique GPC à droite. Le graphe obtenu avec la première a un état de moins.

Approche de Magniette-Pilard-Rozoy L'approche proposée par Magniette, Pilard et Rozoy [MPR03] utilise de la même façon l'indépendance de transitions pour construire des pas. Par

[4]Une réception non spécifiée correspond à l'envoi d'un message non attendu par l'agent récepteur

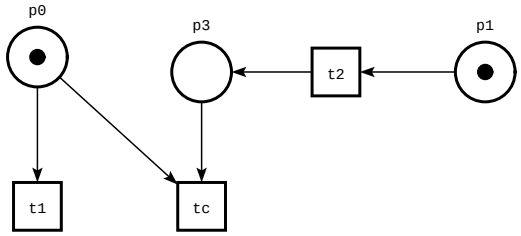


Figure II.15: Réseau de Petri avec confusion

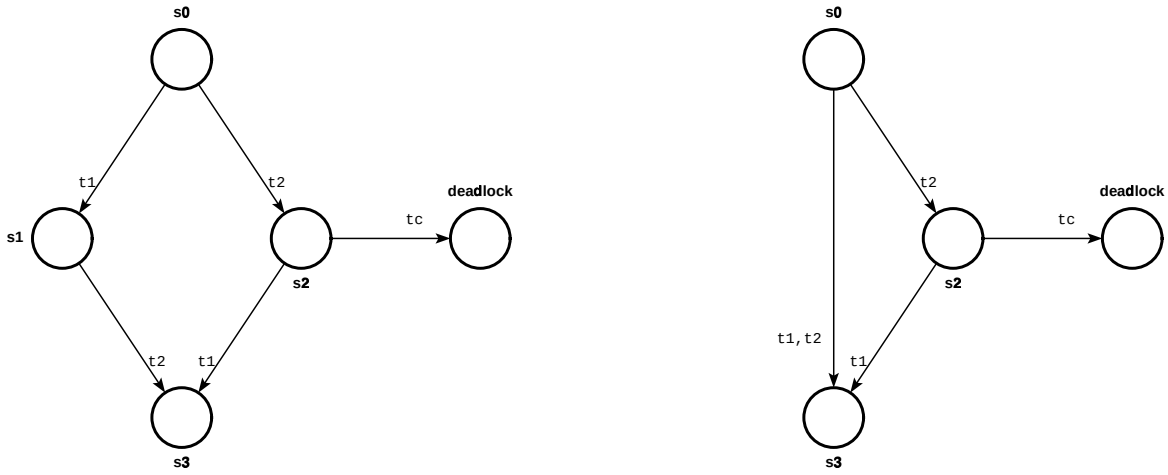


Figure II.16: Cas de confusion : graphe exhaustif et graphe construit par l'approche SRA

contre le cas de confusion est ici géré un peu différemment : au lieu d'éviter le cas de confusion, celui-ci est d'abord ignoré, puis la construction revient en arrière lorsqu'elle découvre qu'il y avait le cas de confusion.

Ce retour en arrière est déclenché grâce à l'enregistrement des transitions qui ne sont pas tirées mais qui étaient « en partie » sensibilisées. Ainsi dans le cas où après le tir du pas, on s'aperçoit que la transition n'est toujours pas sensibilisée, mais que la partie complémentaire de la « partie sensibilisée » précédemment est sensibilisée, alors un retour en arrière dans la construction du graphe de comportement est fait pour explorer les transitions concernées isolément et non plus groupées en pas.

Cette approche nécessite d'une part l'enregistrement des transitions qui antérieurement ont été partiellement sensibilisées et de leur partie sensibilisée, et d'autre part être capable de faire des « retours en arrière » pour finalement explorer les parties du graphe, qui avaient été ignorées par la construction d'un pas.

Par contre cette approche optimiste (on suppose dans un premier temps, que la construction de pas ne pose pas de problèmes) permet de revenir corriger uniquement si le cas de confusion a **effectivement** eu lieu, alors que les techniques SRA et GPC utilisent l'analyse statique pour décider a priori de ne pas faire de pas. En effet lorsque l'analyse statique, détecte qu'il y a éventuellement un problème de cas de confusion dans le futur, les pas ne sont pas construits, alors qu'éventuellement le cas de confusion n'existait pas réellement.

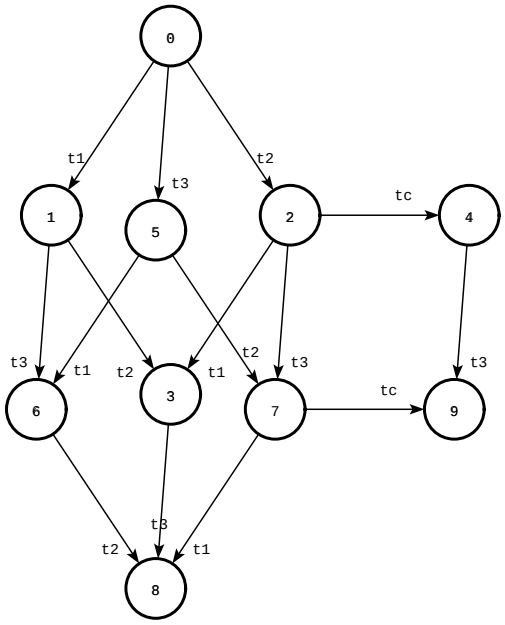


Figure II.17: Cas de confusion+transition : graphe exhaustif

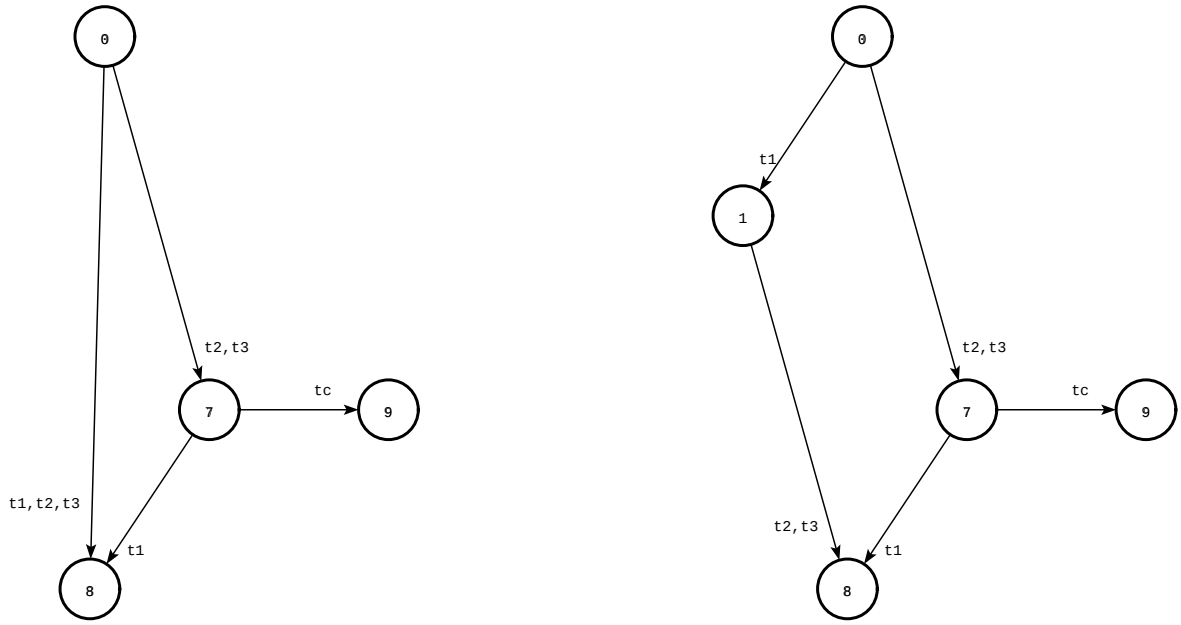


Figure II.18: Cas de confusion+transition : approche SRA et approche GPC

3.3 Préservation de propriétés spécifiques

Des techniques de réduction ont été développées pour construire des graphes préservant des propriétés spécifiques. Nous allons voir différentes techniques qui sont des évolutions, ou des généralisations des techniques vues précédemment.

3.3.1 Techniques par « choix »

Ensembles proviso Les ensembles persistants sont très intéressants pour la préservation des blocages. En effet certaines parties du graphe de comportement du système, inutiles pour la propriété de blocage sont éventuellement complètement ignorées. Cet avantage devient cependant un inconvénient majeur pour la préservation de propriétés plus spécifiques. Ce phénomène est nommé « ignoring problem » [God96]. Pour pallier ce problème, les ensembles proviso ont été proposés. Une exploration par ensembles persistants est basée sur l'intuition qu'une transition qui reste sensibilisée n'a pas besoin d'être explorée immédiatement mais pourra l'être plus tard. L'« ignoring problem » survient lorsque finalement on ne l'explore jamais en ayant indéfiniment retardé son exploration. Pour éviter cette situation on s'interdit de revenir sur un état déjà exploré (fermeture d'un circuit) si des transitions sensibilisées n'ont pas été tirées.

Définition 43 *Ensemble proviso*

Dans un état s avec E l'ensemble des transitions sensibilisées. $P_r \subseteq E$ est un ensemble proviso ssi :

(C_1) P_r est persistant

(C_2) $\forall t \in P_r, s \xrightarrow{t} s_1$ ne doit pas fermer un circuit ou $P_r = E$ (fully expanded)

Cette méthode permet de construire des graphes portant le nom d'« automate trace ». Dans un tel automate à toute séquence d'exécution du système correspond une séquence d'exécution dans l'automate qui lui est équivalente (au sens de Mazurkiewicz) et inversement.

Dans le cas d'une exploration en profondeur d'abord, pour vérifier qu'un ensemble satisfait la propriété (C_2), on compare les états atteints par le tir des transitions avec les états se trouvant sur la pile d'exploration ; si le tir d'une transition de l'ensemble considéré construit un état se trouvant sur la pile d'exploration alors cette transition ferme un circuit.

L'algorithme de la Table II.7 permet la mise en oeuvre des ensembles proviso. Cet algorithme est semblable à l'algorithme des ensembles persistants, cependant l'appel à la fonction construisant un ensemble persistant a été remplacé par l'appel à une fonction construisant un ensemble proviso.

```
develop_Proviso(E):
        Pr ← P_r(E)
        develop_exhaustive(Pr)
```

Table II.7: Algorithme d'exploration par ensembles proviso

Ensembles amples Les ensembles amples ont été définis par Peled [Pel98, CGP00]. Ils constituent une généralisation des ensembles persistants et des ensembles proviso, qui permet la préservation des formules de LTL closes par bégayement. A partir d'une formule LTL et plus particulièrement des propositions atomiques sur lesquelles elle est définie, on définit un ensemble de transitions dites « observables » T_{Obs}. Ces transitions observables permettent de donner un rôle particulier aux transitions importantes par rapport à la propriété à vérifier.

Définition 44 *Ensemble ample*

Soit s un état et E l'ensemble des transitions sensibilisées.
$A \subseteq E$ est un ensemble ample dans s ssi :

(C_1) A est persistant
(C_2) $\forall t \in A, s \overset{t}{\twoheadrightarrow} s_1$ ne doit pas fermer un circuit ou $A = E$ (fully expanded)
(C_3) $A \cap T_{Obs} = \emptyset$ ou $A = E$ (fully expanded)

Cette définition nous permet de montrer aisément que les ensembles amples généralisent les ensembles persistants et les ensembles proviso. Si uniquement la condition (C_1) est vérifiée, alors la technique des ensembles amples correspond à la technique des ensembles persistants et permet donc uniquement la préservation des états de blocage. Avec les conditions (C_1) et (C_2) les ensembles amples sont des ensembles proviso. Lorsque la condition (C_3) est vérifiée, on a les propriétés suivantes :

Propriété 6 *Dans le graphe réduit construit avec les ensembles amples, dans tout circuit il y a au moins un noeud « fully expanded ».*

Cette propriété peut être rapprochée de la condition (C_1) qui empêche qu'une transition « laissée de côté temporairement » soit désensibilisée par une transition tirée et par conséquent on a la propriété suivante :

Propriété 7 *Si une transition t est sensibilisée dans un état s et que cet état appartient à un circuit, alors il existe un état de ce circuit à partir duquel cette transition est tirée.*

Cette propriété permet de s'affranchir de l'« ignoring problem » présenté dans le cadre des proviso : une transition sensibilisée ne peut pas indéfiniment être délayée dans le graphe réduit. En utilisant ces différentes propriétés on montre que les ensembles amples permettent de vérifier des propriétés de la logique linéaire bégayement invariantes.

3.3.2 Techniques par « Pas »

L'approche *GPC* présentée dans sa formule de base ne préserve pas uniquement les blocages, mais également des propriétés générales d'accessibilité telles que la vivacité [VAM96]. La méthode peut également être spécialisée pour vérifier certaines propriétés telles que la bisimulation faible [VAM96] ou les sémantiques de refus [VM97].

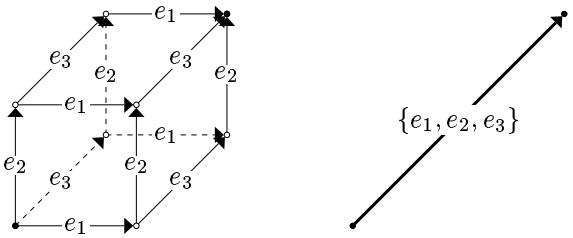
Les dernières extensions de l'approche *SRA* [KT00] permettent l'utilisation de cette méthode dans un cadre plus général en particulier pour la vérification de formules de LTL bégayement invariantes.

Chapter III

Graphe de Pas couvrants

La vérification formelle d'un système consiste à considérer tous les comportements possibles du système. Aussi si l'on construit tous ces comportements, lorsque plusieurs transitions indépendantes peuvent être exécutées en parallèle dans le système, on construit tous les entrelacements possibles de ces transitions.

L'approche des graphes de pas couvrants, introduite dans [VAM96, VM97], consiste à ne pas considérer tous les entrelacements possibles en construisant un « pas » (lorsque cela est possible) permettant d'atteindre directement l'état final qui aurait été atteint par chacun des entrelacements. La figure III.1 illustre le bénéfice obtenu dans le cas de la dérivation de n événements parallèles. L'exploration exhaustive donne lieu à un hypercube d'ordre n : on obtient un nombre exponentiel d'arcs et d'états. L'approche par pas fournit un résultat optimal puisque le nombre d'états (d'arcs) est indépendant du nombre d'événements.



Combinatoire	Exhaustive	**Pas**
Etats	2^n	2
Arcs	$n \times 2^{n-1}$	1

Figure III.1: Dérivation de 3 événements indépendants

Les graphes de pas couvrants préservent les états de blocage et les traces maximales. Différentes versions ont également été proposées pour préserver l'équivalence observationnelle, ou la sémantique de refus.

Dans ce chapitre nous présentons les graphes de pas couvrants généraux. Les graphes de pas couvrant sont définis dénotationnellement par rapport à la relation de préservation existant entre un graphe complet et son graphe de pas couvrants associé. Cette approche permet de définir un graphe de pas couvrant indépendamment de la façon dont on va construire le graphe et d'être proche de la structure et des propriétés voulues du graphe construit. Nous donnons alors un algorithme paramétré (par des fonctions) permettant de construire un graphe de pas couvrant.

Nous donnons enfin des conditions suffisantes sur les fonctions paramétrées pour que l'algorithme construise effectivement un graphe de pas couvrant. Nous démontrons alors que ces conditions sont effectivement suffisantes. Enfin nous donnons un exemple d'instanciation de ces fonctions qui vérifie les conditions suffisantes ; cet exemple est appelé « graphe de conflits croisés ».

1 Groupement des transitions en pas

1.1 Les pas de transitions

La caractéristique principale d'un pas de transitions est que si toutes les transitions sont exécutables, alors elles le sont dans n'importe quel ordre. Pour cela il est donc nécessaire que toutes les transitions du pas soient indépendantes.

Définition 45 *Pas de transitions*
Un ensemble π de transitions définit un « pas de transitions » par rapport à $\wr$ ssi $\forall t_1, t_2 \in \pi$: $t_1 \wr t_2$.
L'ensemble des « pas de transitions » formés sur T par rapport à $\wr$ est noté $Step(T, \wr)$.Lorsque le contexte le permet, on note $Step(T)$ à la place de $Step(T, \wr)$.

D'après cette définition, il est évident que tout sous-ensemble d'un pas, est également un pas, d'où la propriété suivante :

Propriété 8

$$\forall \pi \in Step(T, \wr), \pi' \subseteq \pi \Rightarrow \pi' \in Step(T, \wr)$$

1.2 Linéarisation et traces d'un pas

Les transitions d'un pas doivent pouvoir être exécutées dans n'importe quel ordre, on parle de linéarisation d'un pas : une séquence de transitions est une linéarisation d'un pas si le pas et la séquence ont exactement les mêmes transitions. On définit par la fonction Seq l'ensemble de toutes les linéarisations possibles d'un pas.

Définition 46 *Ensemble des séquences associées à un pas de transitions*
Soit Seq l'application de $\mathcal{P}(T) \mapsto \mathcal{P}(T^*)$ définie par :
$Seq(\emptyset) = \{\epsilon\}$
$Seq(\pi) = \bigcup_{t \in \pi} Seq(\pi \setminus \{t\}) \otimes t$
où pour $\Omega \in \mathcal{P}(T^*)$ et $w \in T^*$: $\Omega \otimes w = \{\omega.w \ : \omega \in \Omega\}$

Un pas de k transitions à un nombre de linéarisation égal à factorielle k.

$$Card(Seq(\pi)) = Card(\pi)!$$

La taille de cet ensemble met en évidence le gain obtenu par la construction d'un pas. Mais le gain est encore plus important puisque d'autres séquences auraient été construites à partir des états intermédiaires.

Propriété 9 *Soit $\pi \in Step(T, \wr)$*

$$w_1 \text{ et } w_2 \in Seq(\pi) \Rightarrow [w_1]_{(T, \wr^C)} = [w_2]_{(T, \wr^C)}$$

Démonstration

La démonstration de cette propriété est évidente puisque les transitions d'un pas sont indépendantes, donc les deux séquences w_1 et w_2 appartiennent à la même classe de Mazurkiewicz.

Puisque toutes les linéarisations d'un pas ont la même trace, on peut définir aisément la trace d'un pas de transitions comme la trace de l'une de ses séquences.

Définition 47 *Trace d'un pas de transitions*

Soit $\pi \in Step(T, \wr)$, on définit alors

$$[\pi]_{(T,\wr^C)} = [\mathcal{E}]_{(T,\wr^C)} \text{ où } \mathcal{E} \in Seq(\pi)$$

De même que pour les séquences de transitions, **la trace d'une séquence de pas** est définie par la trace d'un mot obtenu par concaténation des mots associés à chacun des pas de la séquence : Pour $\pi_1.\pi_2 \ldots \pi_n \in Step(T, \wr)^*$, $[\pi_1.\pi_2 \ldots \pi_n]_{(T,\wr^C)} = [\mathcal{E}_1.\mathcal{E}_2 \ldots \mathcal{E}_n]_{(T,\wr^C)}$ où $\mathcal{E}_i \in Seq(\pi_i)$

1.3 Extension de la relation d'accessibilité aux pas de transitions

On étend la propriéte de sensibilisation d'une transition à la sensibilisation d'un pas. Un pas est sensibilisé si et seulement si toutes les linéarisations de ce pas sont tirables.

Définition 48 *Sensibilisation d'un pas*

Soit $\twoheadrightarrow$, l'extension de $\rightarrow$ aux pas de transitions.

$\forall s, s' \in S, \forall \pi \in Step(T)$:

$$s \overset{\pi}{\twoheadrightarrow} s' \text{ ssi } \forall \omega \in Seq(\pi) s \overset{\omega}{\Rightarrow} s'$$

Dans la pratique vérifier si toutes les linéarisations sont exécutables reviendrait à construire comme nous l'avons vu les « factorielle k » combinaisons possibles. En fait comme toutes les transitions à l'intérieur d'un pas sont indépendantes, il suffit de vérifier que chacune d'elles est sensibilisée dans l'état depuis lequel on veut tirer le pas.

Propriété 10 *Condition de sensibilisation d'un pas*

$$s \overset{\pi}{\twoheadrightarrow} \text{ ssi } \forall t \in \pi : s \overset{t}{\longrightarrow}$$

1.4 Extensions de $\wr$ aux séquences de transitions et aux pas de transitions

Définition 49 *Extensions de $\wr$*

Soit $\wr \subset Step(T) \times Step(T)$ définie par $\pi_1 \wr \pi_2$ ssi $\pi_1 \times \pi_2 \subset \wr$

Soit $\wr \subset Step(T) \times T^*$ définie par $\pi_1 \wr w$ ssi $\pi_1 \times ||w|| \subset \wr$

Soit $\wr \subset T^* \times T^*$ définie par $w_1 \wr w_2$ ssi $||w_1|| \times ||w_2|| \subset \wr$

Propriété 11 *Propriétés du Losange*

1. *Pour $t \in T, w \in T^*$: Si $s \overset{t}{\longrightarrow}, s \overset{w}{\Rightarrow}$ et $t \wr w$ alors $s \overset{t.w}{\Rightarrow} s'$ et $s \overset{w.t}{\Rightarrow} s'$*
2. *Pour $w_1, w_2 \in T^*$: Si $s \overset{w_1}{\Rightarrow}, s \overset{w_2}{\Rightarrow}$ et $w_1 \wr w_2$ alors $s \overset{w_1.w_2}{\Rightarrow} s'$ et $s \overset{w_2.w_1}{\Rightarrow} s'$*
3. *Pour $w \in T^*, E \in Step(T)$: Si $s \overset{w}{\Rightarrow}, s \overset{E}{\twoheadrightarrow}$ et $E \wr w$ alors $s \overset{\mathcal{E}.w}{\Rightarrow} s'$ et $s \overset{w.\mathcal{E}}{\Rightarrow} s'$ $\forall \mathcal{E} \in Seq(E)$*
4. *Pour $E_1, E_2 \in Step(T)$: Si $s \overset{E_1}{\twoheadrightarrow}, s \overset{E_2}{\twoheadrightarrow}$ et $E_1 \wr E_2$ alors $s \overset{E_1 \cup E_2}{\twoheadrightarrow}$*
   Et $s \overset{\mathcal{E}_1.\mathcal{E}_2}{\Rightarrow} s'$ et $s \overset{\mathcal{E}_2.\mathcal{E}_1}{\Rightarrow} s'$ $\forall \mathcal{E}_i \in Seq(E_i)$

Démonstration

1. par récurrence sur $|w|$.
2. par récurrence sur l'un des $|w_i|$ et propriété (1).
3. est une conséquence directe de (2).
4. est une conséquence directe de (3).

2 Graphe de pas couvrants

2.1 Graphe de pas couvrants (définition)

Définition 50 *Graphe de Pas Couvrant*
Soit $\Sigma = \langle S, s_o, T, \rightarrow \rangle$ *un STE,* $\wr$ *une relation d'indépendance sur* Σ *et* $\# = \wr^C$.
$\Sigma_R = \langle S_R, s_o, T_R, \rightarrow_R \rangle$ *est un* **Graphe de Pas Couvrant** *pour* Σ, *par rapport à* $\wr$, *ssi*
(1) $S_R \subseteq S$
(2) $T_R \subseteq Step(T, \wr)$
(3) $\forall s, s' \in S \cap S_R, \forall \pi \in T_R \ : s \xrightarrow{\pi}_R s' \ \ implique \ \ s \overset{w}{\twoheadrightarrow} s' \quad \forall w \in Seq(\pi)$
(4) $\forall s \in S \cap S_R, \forall s' \in S \ :$

$$s \overset{w}{\longrightarrow} s' \ \ implique \ \begin{cases} \exists s" \in S_R, \exists w' \in T^*, \exists P_{w.w'} \in {T_R}^* \ : \\ s' \overset{w'}{\longrightarrow} s", s \overset{P_{w.w'}}{\rightarrow}_R s" \ \ et \ [w.w']_{(T,\#)} = [P_{w.w'}]_{(T,\#)} \end{cases}$$

Notons que tout graphe d'accessibilité peut être considéré comme son propre graphe de pas couvrants (en prenant $\wr = \emptyset$).

Nous donnons un exemple de graphe de pas couvrant sur l'exemple du scheduler de Milner présenté en début de chapitre II. La Figure III.2 représente le graphe de pas couvrants obtenu dans le cas de 2 sites.

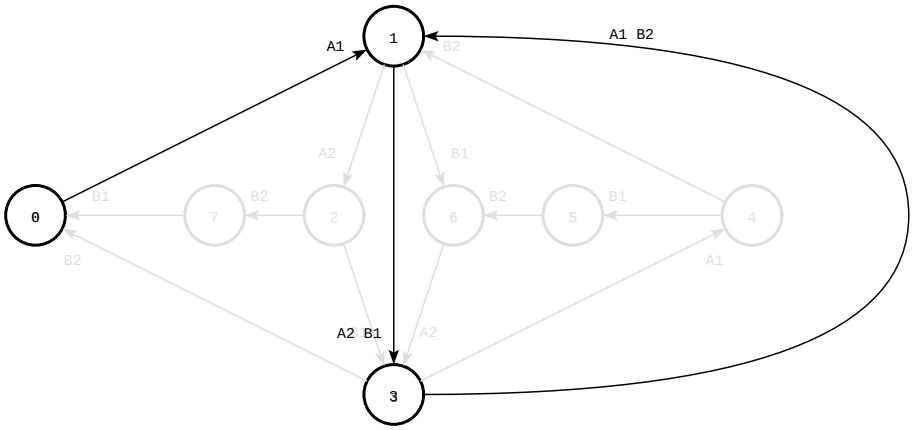


Figure III.2: Scheduler de Milner - GPC pour $n = 2$

La condition (1), impose que tout état du GPC soit un état du STE. Pour cet exemple, $t_1 \wr t_2$ ssi $t_1 \neq t_2$, en conséquence chaque sous-ensemble de T constitue un pas de transitions et (2) est satisfaite. (3) signifie qu'à chaque pas du graphe de pas correspond toutes les linéarisations dans le STE, ainsi : $2 \overset{\{A2,B1\}}{\rightarrow}_R 6$ correspond à $2 \overset{A2.B1}{\Rightarrow} 6$ et $2 \overset{B1.A2}{\Rightarrow} 6$. Finalement, la condition (4)

exprime une condition de « couverture » entre les séquences du STE et celles associées au GPC. Ainsi pour S, on a $1 \stackrel{a_1.a_2.b_2.b_1}{\Rightarrow} 1$, la condition 4 assure l'existence dans Σ_R d'une séquence de pas la couvrant. Si on étend la séquence $a_1.a_2.b_2.b_1$ par la transition a_1, elle est alors couverte dans le GPC par la séquence suivante $\Pi = \{a_1\}.\{a_2, b_1\}.\{a_1, b_2\}$ $: 1 \stackrel{\Pi}{\rightarrow}_R 2$ et $1 \stackrel{a_1.a_2.b_2.b_1.a_1}{\Rightarrow} 2$ avec $[a_1.a_2.b_2.b_1.a_1]_{(T,\wr^C)} = [\{a_1\}.\{a_2, b_1\}.\{a_1, b_2\}]_{(T,\wr^C)}$

2.2 Propriétés préservées par les graphes de pas couvrants

L'intérêt d'une technique de réduction, au-delà du facteur de compression qu'elle offre effectivement, réside dans la capacité d'analyse qu'elle procure. La technique d'« exploration par pas » est une technique assez générale qui peut être déclinée suivant le type de propriétés que l'on souhaite vérifier. La version classique des GPC préserve les états de **blocage** et de **vivacité**. Nous rappelons ici les définitions et propriétés qui sont démontrées dans [VM97].

Définissons l'ensemble des blocages d'un système de transitions étiquetées :

$$Deadlock(\Sigma) = \{s \in S \ : s \stackrel{t}{\not\rightarrow} \quad \forall t \in T\}$$

Propriété 12 *Préservation des états de blocage*
Si Σ_R est un GPC de Σ alors

$$Deadlock(\Sigma) = Deadlock(\Sigma_R)$$

Un système est vivant pour un ensemble T' de transitions, si toutes les transitions de cet ensemble peuvent toujours être exécutées dans le futur.

Définition 51 *T'-vivacité d'un système*
Soit $\Sigma = \langle S, s_o, T, \rightarrow \rangle$ un système de transitions étiquetées. Soit $T' \subseteq T$ un sous-ensemble de transitions.

$$\Sigma \text{ est } T'\text{-vivant ssi } \forall s \in S, \forall t \in T', \exists w \in T^* \text{ tel que } s \stackrel{w.t}{\Longrightarrow}$$

Pour généraliser cette définition aux graphes de pas couvrants, une transition est vivante dans un GPC, si on peut depuis tout état, atteindre un état dans lequel un pas contenant la transition est sensibilisé.

Définition 52 *T'-vivacité d'un GPC*
Soit $\Sigma_R = \langle S_R, s_o, T_R, \rightarrow_R \rangle$ un GPC. Soit $T' \subseteq T$ un sous-ensemble de transitions.

$$\Sigma_R \text{ est } T'\text{-vivant ssi } \forall s \in S_R, \forall t \in T', \exists \Pi \in {T_R}^*, \exists \pi \in T_R \text{ tel que } s \stackrel{\Pi.\pi}{\rightarrow}_R \text{ avec } t \in \pi$$

Propriété 13
Si Σ_R est un GPC pour Σ alors Σ_R est T'-vivant ssi Σ est T'-vivant.

2.3 Algorithme de construction d'un graphe de pas couvrants

Le graphe de pas couvrants doit effectivement être construit uniquement à partir de la description initiale et non pas à partir du graphe complet, puisque l'objectif est justement de ne pas construire celui-ci.

Cette section introduit un schéma général d'algorithme pour la construction de graphes de pas couvrants. Les conditions générales à satisfaire pour assurer la correction de l'algorithme sont présentées. Finalement, une instance particulière de ce schéma général est proposée en section 5.

La table III.1 présente l'algorithme de construction d'un graphe de pas couvrants. L'algorithme est paramétré par une relation d'indépendance $\wr$. Les transitions sensibilisées sont partitionnées en deux sous-ensembles grâce aux fonctions T_U et T_M. L'ensemble T_u contient les transitions à explorer d'une façon standard et T_m contient celles dont l'exploration sera menée par des pas. Ces dernières seront nommées « fusionnables » dans la suite. Π_{T_M} est l'ensemble des pas de transitions, construit par la fonction Π.

```
develop_GPC(E):
        T_u ← T_U(E, ≀)
        T_m ← T_M(E, ≀)
        develop_exhaustive(T_u)
        Π_{T_m} ← Π(T_m, #)
        develop_by_step(Π_{T_m})
```

```
develop_by_step(Π):
    for each π in Π do
        s' ← step_fire(s, π)
        G ← G ∪ {< s, π, s' >}
        if s' ∉ S then
            S ← S ∪ {s'}
            Ajouter(Tmp, s')
```

Table III.1: Algorithme de construction d'un *GPC*

La fonction $develop\_by\_step(\Pi)$, définie dans la table III.1, est similaire à la fonction $develop\_exhaustive(E)$, excepté le fait que les états intermédiaires ne sont pas enregistrés ; toutes les transitions du pas sont tirées par la fonction $step\_fire()$ et seul l'état final est ajouté au graphe.

3 Conditions Suffisantes pour obtenir un GPC

3.1 Conditions Suffisantes

La table III.2 présente les conditions générales sur les ensembles T_u, T_m et Π_{T_m} assurant l'exactitude de l'algorithme de la table III.1. Ces conditions sont paramétrées par une relation de conflit # transitive.

$\forall s \in S$:

(CA_1) $T_m \cup T_u = Enabled(s)$ et $T_m \cap T_u = \emptyset$

(CA_2) Si $t \in T_m$ alors $t' \# t \Rightarrow t' \in Enabled(s)$

* * * * *

(CB_1) $\forall \pi \in \Pi_{T_m} : \pi \in Step(Enabled(s), \#^C)$

(CB_2) $\forall P \in Step(Enabled(s), \#^C), \exists \pi \in \Pi_{T_m} : P \subseteq \pi$

Table III.2: Conditions suffisantes sur T_u, T_m et Π_{T_m}

Les conditions CA_1 et CA_2 sont respectivement associées aux ensembles T_u et T_m :

- CA_1 impose que T_u et T_m forment une partition de l'ensemble des transitions sensibilisées dans l'état s ($Enabled(s)$),
- CA_2 assure que toute transition en conflit avec une transition fusionnable est elle-même sensibilisée.

Les conditions CB_1, CB_2 concernent l'ensemble des pas de transitions Π_{T_m} :

- CB_1 assure que les pas de transitions du graphe couvrant sont aussi des pas pour la relation $\#\!\!\#$,
- CB_2 assure que tout pas de transition pour $\#\!\!\#$ est couvert par un pas de transition appartenant à Π_{T_m}.

Les conditions CA_2, CB_1 d'une part et CB_2 d'autre part, sont illustrées dans les paragraphes 3.2.1 et 3.2.2.

Notons qu'une erreur s'était glissée dans l'article [VAM96] ; la fermeture transitive de la relation de conflit $\#\!\!\#$ avait été remplacée par la relation de conflit initiale $\#$. Nous remercions Jean Fanchon de nous l'avoir signalée. La fermeture transitive $\#\!\!\#$ est plus faible que la relation de conflit initiale $\#$.

3.2 Justification du choix de ces conditions

3.2.1 Situations de Confusion

Pour justifier les conditions CA_2 et CB_1, nous considérons à nouveau le réseau de Petri avec le cas de confusion de la Figure II.10. On appelle S son graphe d'accessibilité.

Motivation de CA_2 : on considère le STE de la figure III.3 comme graphe de pas pour le réseau de Petri de la figure II.10. A partir de l'état 1 le pas de transitions $\{A1, B2\}$ est constitué alors que la transition $A1$ est en conflit avec la transition D qui n'est pas sensibilisée. L'évolution « atomique » associée au pas de calcul $\{A_1, B_2\}$ *élimine* les états 2 et 3. Or l'état 3 sensibilise la transition D qui conduit à un blocage. La réalisation atomique du pas {A1,B2} éliminant l'état 3, élimine aussi la possibilité d'occurrence de la transition D et l'état de blocage associé. Dans cette situation, la condition de « couverture » (Définition 50) est violée.

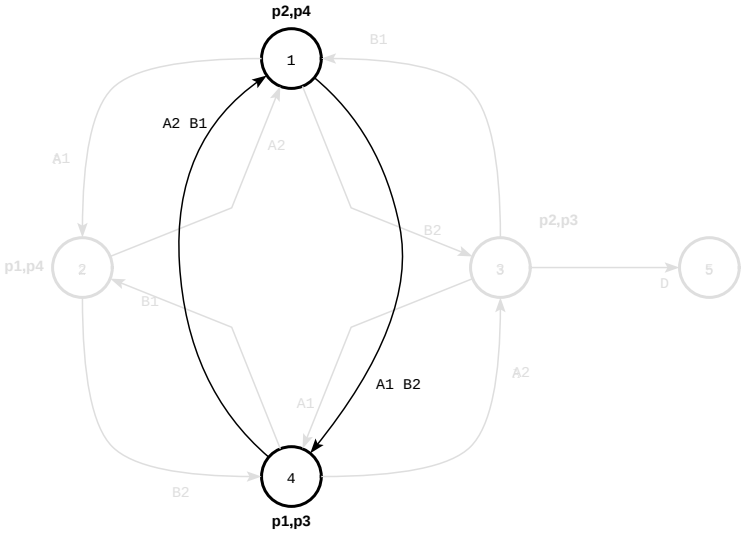


Figure III.3: Situation de confusion - Motivation de CA_2

Motivation de CB_1 : on considère le STE de la Figure III.4 appelé $S2$. A partir de l'état 3, le pas {A1,B1} est constitué alors que $A1$ et $B1$ sont en « conflit indirect » via D. (i.e. $\{A1, B1\} \in Step(Enabled(s_3), \wr)$ mais $\{A1, B1\} \notin Step(Enabled(s_3), \#^C)$ car $A1\#D$ et $B1\#D$, d'où $A1 \# B1$).

La séquence $1 \overset{B2.B1.B2.D}{\Rightarrow} 5$ de S (cf. Figure II.10) n'est pas couverte dans $S2$. Comme l'état 5 est un blocage, une séquence de l'état 1 à l'état 5 doit exister dans le graphe de pas $S2$. La seule séquence de longueur 4 dans $S2$ est[1] $1 \overset{\{A1\}.\{B2,A2\}.\{D\}}{\rightarrow_R} 5$ or $[\{A1\}.\{B2, A2\}.\{D\}] \neq [B2.B1.B2.D]$.

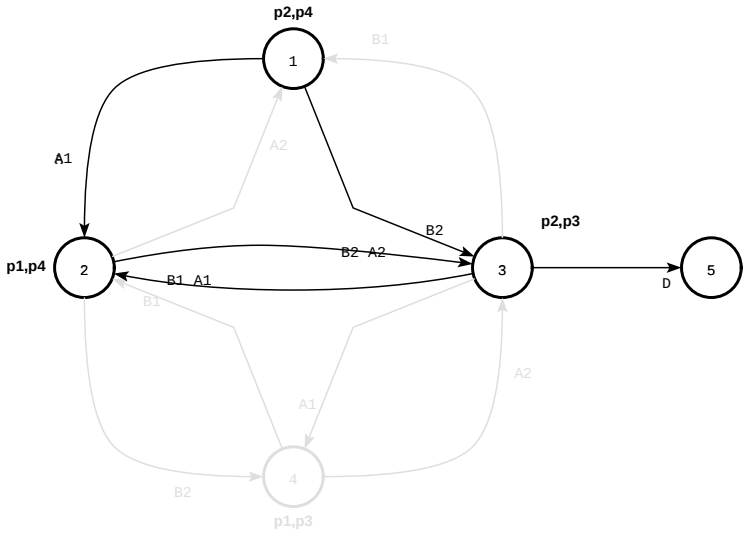


Figure III.4: Situation de confusion - Motivation de CB_1

Finalement le graphe de la figure III.5 respectant CA_2 et CB_1 est un GPC valide pour le réseau de Petri de la figure II.10.

3.2.2 Couverture de Pas

Le réseau de la figure III.6 présente deux « ensembles de conflits indépendants », S est le STE associé à N. La figure III.7 propose des graphes de pas : S_2 est un GPC de S, tandis que S_1 viole la condition CB_2. En effet si on considère la séquence $0 \overset{t0.t2}{\Rightarrow} 2$ de S, celle-ci doit être couverte dans tout GPC. Ce n'est pas le cas pour $S1$, on a : $0 \overset{\{t0,t3\}}{\rightarrow_R} 2$ et $0 \overset{\{t1,t2\}}{\rightarrow_R} 2$ mais $[\{t0, t3\}] \neq [t0.t2]$ et $[\{t1, t2\}] \neq [t0.t2]$

3.3 Démonstration que les Conditions sont Suffisantes pour la génération d'un GPC

On doit montrer que le schéma d'algorithme de la table III.1, sous réserve que les conditions suffisantes de la table III.2 soient satisfaites, fournit effectivement un GPC (Définition 50). Pour cela démontrons les quatre points de la définition.

Preuve des points (1), (2) et (3) (Définition 50)

[1] L'égalité de traces assure que les deux séquences ont la même longueur

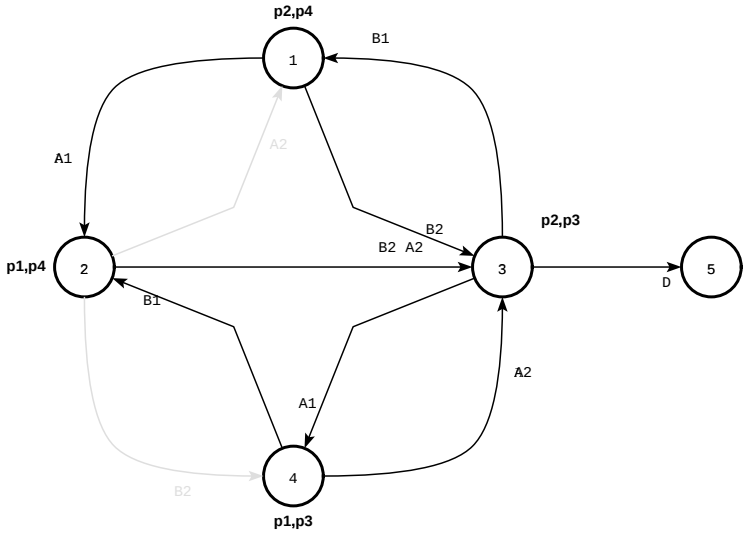


Figure III.5: Situation de confusion - GPC

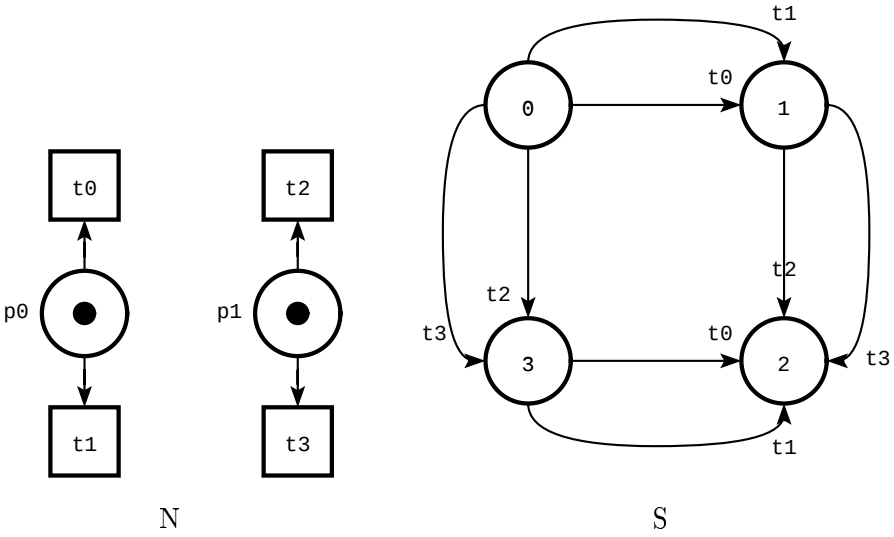


Figure III.6: Illustration de CB_2 - Modèle et Exploration exhaustive

Les points (2) et (3) de la définition d'un graphe de pas couvrants sont des conséquences directes des conditions CA_1, CB_1 et du fait que la relation $\#\!\#^C$ soit un sous-ensemble de $\wr$: Lorsque $s \xrightarrow[R]{E} s'$, deux cas sont à considérer :

- Soit $E = \{t\}$ $(t \in T_U)$ et alors $s \xrightarrow{t} s'$ (donc (3)) et $\{t\} \in Step(T, \wr)$ (donc (2))
- Soit $E \in \Pi_{T_M}$, $E = \{t_1, t_2, \ldots t_k\}$ et alors d'après CB_1, $E \in Step(Enabled(s), \wr)$ (donc (2)) et $s \overset{E}{\twoheadrightarrow} s'$ d'après la propriété 10 sur les pas de transition.

L'énumération débute avec $H = \{s_0\}$ $(s_0 \in S)$, (2) et (3) assurent que tout état obtenu à partir d'un état de H appartient effectivement à S, la condition (1) est donc satisfaite.

Le preuve de la condition (4) est faite dans la section suivante.

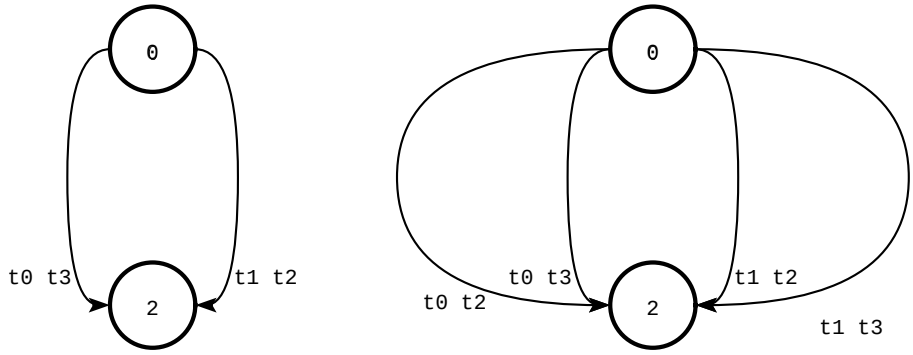


Figure III.7: Illustration de CB_2 - Graphe de pas couvrants

4 Démonstration que les conditions suffisantes assurent toujours la construction d'un GPC

La démonstration complète n'a jamais été publiée. Nous utilisons l'opportunité de la place offerte par la thèse pour rédiger complètement la démonstration.

Montrons que la condition suffisante (CB_2) implique toujours la propriété (4) des graphes de pas couvrants. On fait une preuve par récurrence sur la longueur de w, en s'appuyant sur un lemme de normalisation, qui est prouvé en introduisant une application de normalisation. Nous allons détailler chacune de ces étapes.

4.1 Application de normalisation

On introduit un opérateur de normalisation, π, qui permet d'extraire de toute séquence de transitions w sensibilisée dans un état s, un pas maximal ou un préfixe maximal d'un tel pas.

Soit $\pi : S \times Step(T) \times T^* \times T^* \mapsto Step(T) \times T^*$ défini par :

$\pi(s, E, w, \epsilon) = \quad (E, w)$

$\pi(s, E, w_1, w_2) = \quad (E, w_1.w_2) \quad$ si $E \in \Pi_{T_m}$

$$\pi(s, E, w_1, t.w') = \begin{cases} \pi(s, E \cup \{t\}, w_1, w') & \text{si } t \in T_m, \\ & t \wr w_1 \text{ et } E \cup \{t\} \in Step(T, \#^C, s) \text{ et } t \notin E \\ (\{t\}, \mathcal{E}.w_1.w') & \text{si } t \in T_u, \\ & t \wr w_1 \text{ et } \mathcal{E} \in Seq(E) \\ \pi(s, E, w_1.t, w') & \text{sinon} \end{cases}$$

4.1.1 Exemples d'utilisation de l'application de normalisation

Pour illustrer l'intuition donnée en introduction de l'application de normalisation, nous allons l'utiliser sur deux exemples. Pour cela considérons le réseau de Petri de la Figure II.11 dont le graphe de pas couvrants a été donné en figure II.14.

Considérons la séquence $w = t_3.t_2.t_1.t_4$ et appliquons dessus l'application de normalisation.

$\pi(s_0, \emptyset, \epsilon, t_3.t_2.t_1.t_4) = \pi(s_0, \{t_3\}, \epsilon, t_2.t_1.t_4)$ { Car $t_3 \in T_m$ et les conditions sont vérifiées
$= \pi(s_0, \{t_3, t_2\}, \epsilon, t_1.t_4)$ { car $t_2 \in T_m$...
$= \pi(s_0, \{t_2, t_3\}, t_1, t_4)$ { car t_1 n'est pas sensibilisée dans l'état s_0 donc cas du « sinon »
$= \pi(s_0, \{t_2, t_3, t_4\}, t_1, \epsilon)$ { car $t_4 \in T_m$...
$= (\{t_2, t_3, t_4\}, t_1)$ { cas de terminaison lorsque $w_2 = \epsilon$.

On a donc bien extrait un « bout » (ou préfixe) de pas le plus grand possible par rapport à la séquence w. On peut effectivement prolongée la séquence pour utiliser un pas contenant l'ensemble $\{t_2, t_3, t_4\}$, par exemple le pas $\{t_2, t_3, t_4, t_6\}$. Notons que ce pas n'est pas unique. Si la séquence était plus longue, l'application récursive sur la suite de la séquence permet d'obtenir une séquence de pas couvrant la séquence considérée.

Comme second exemple considérons la séquence $w = t_2.t_0.t_6$. Cette séquence est intéressante car elle contient la transition t_0 qui est sensibilisée dans l'état initial mais en conflit avec une transition non sensibilisée (donc $t_0 \in T_u$).

$\pi(s_0, \emptyset, \epsilon, t_2.t_0.t_6) = \pi(s_0, \{t_2\}, \epsilon, t_0.t_6)$ { Car $t_2 \in T_m$...
$= (\{t_0\}, t_2.t_6)$ { Car $t_0 \in T_u$ et dans ce cas on isole la transition

L'application de normalisation retourne donc la transition appartenant à l'ensemble T_u, et permet d'isoler d'abord cette transition pour permettre la construction d'un pas sur la suite.

4.1.2 Propriétés de l'application de normalisation π

Propriété 14 *Propriétés de l'application de normalisation π*

Soit $s \in S$ et $w, w_1 \in T^*$ tels que $s \xrightarrow{w} s'$ et $\pi(s, \emptyset, \epsilon, w) = (E, w_1)$ Alors (1) et (2)

(1) $s \xrightarrow{E} \gg s_1, s_1 \xrightarrow{w_1} s'$ *avec* $[w] = [E.w_1]$

(2) $\forall F \in \Pi_{T_M}$ *tel que* $E \subset F$ *Alors* $F \setminus E \wr w_1$

Démonstration Démonstration des propriétés de π

La preuve du (1) est immédiate. Démontrons la seconde propriété :
Soit $F \in \Pi_{T_M}$ tel que $E \subset F$. Soit R tel que $E \cup R = F$. Montrons que $R \wr w_1$.
Soit $t \in w_1$

Si $s \xrightarrow{t} \not\rightarrow$, comme $F \subset T_m$ d'après la propriété (CA_2) on a $t \wr F$ d'où $t \wr R$. Donc dans la suite nous nous placerons dans le cas où la transition t est sensibilisée : $s \xrightarrow{t}$ avec $t \in T_m$ ou $t \in T_u$

A) Cas où $t \in T_m$

D'après la définition de π, on avait : soit $\neg(t \wr w')$ (avec w' préfixe de w_1), soit $E \cup \{t\} \notin Step(T, \#^C, s)$, soit $t \in E$. Etudions ces trois cas :

1. $\neg(t \wr w')$ avec $w_1 = w'.t.w''$

   Donc $w_1 = w_{11}.t_{conflit}.w_{12}.t.w''$ avec $t \# t_{conflit}$ et $t_{conflit} \wr w_{11}$

   La transition $t_{conflit}$ peut-être sensibilisée ou pas :

   - $s \xrightarrow{t_{conflit}} \not\rightarrow$ Supposons $\exists t_r \in R$ telle que $t_r \# t$ de plus $t \# t_{conflit}$ d'où $t_r \in T_m$, $t_r \# t_{conflit}$ et $s \xrightarrow{t_{conflit}} \not\rightarrow$, donc impossible d'après (CA_2).

- $s \xrightarrow{t_{conflit}}$ donc d'après la définition de π on avait $E' \cup \{t_{conflit}\} \notin Step(T, \#^C, s)$ ou $t_{conflit} \in E'$ (car $t_{conflit} \wr w_{11}$) avec $E' \subset E$. Donc soit $t_{conflit}$ était en conflit avec une transition de E' différente de $t_{conflit}$, soit $t_{conflit}$ apparaissait déjà dans E' :
  - $\exists t_e \in E', t_e \neq t_{conflit}$ telle que $t_{conflit} \# t_e$. Supposons $\exists t_r \in R$ telle que $t_r \# t$ or $t \# t_{conflit}$ d'où $t_r \# t_e$, or $t_r \in F$, $t_e \in F$ et $t_r \neq t_e$, donc impossible d'après (CB_1).
  - $t_{conflit} \in E'$ donc $t_{conflit} \notin R$ or $\forall t_1, t_2 \in F, t_1 \neq t_2 \ : t_1 \wr t_2$ d'où $\forall t' \in R, t_{conflit} \wr t'$ d'où $\forall t' \in R, t \wr t'$ donc $t \wr R$

2. $E \cup \{t\} \notin Step(T, \#^C, s)$

   Donc t est en conflit avec une transition de E différente de t. $\exists t_e \in E, t_e \neq t$ telle que $t_e \# t$. Supposons $\exists t_r \in R$ telle que $t_r \# t$ alors $t_r \# t_e$ or $t_r \in F$ et $t_e \in F$ impossible d'après (CB_1)

3. $t \in E$

   Donc $t \notin R$ or $\forall t_1, t_2 \in F, t_1 \neq t_2 \ : t_1 \wr t_2$ d'où $\forall t' \in R, t \wr t'$ donc $t \wr R$

B) Cas où $t \in T_U$ Supposons qu'il existe $t' \in R$ telle que $t' \# t$, or $s \xrightarrow{t'}$ (car $t' \in R$), donc d'après CA_2, $s \xrightarrow{t}$, sinon t' n'existe pas.
On a $s \xrightarrow{t}$ avec $t \in T_u$, donc $\neg(t \wr w')$ avec $w_1 = w'.t.w''$ car sinon le second cas de la définition de π aurait été utilisé et on aurait eu $E = \{t\}$.
Donc $w_1 = w_{11}.t_{conflit}.w_{12}.t.w''$ avec $t \# t_{conflit}$ et $t_{conflit} \wr w_{11}$.
La transition $t_{conflit} \in T_m$ car si on avait $t_{conflit} \in T_u$, comme $t_{conflit} \wr w_{11}$, le second cas de la définition de π aurait été utilisé. On se retrouve donc dans la même configuration que le (1.) du cas où $t \in T_m$, la démonstration est identique.

4.2 Lemme de normalisation

Nous pouvons maintenant utiliser les propriétés de π pour démontrer le lemme de normalisation suivant :

Lemme 1 *Lemme de normalisation*
$\forall s \in S_R, \forall w \in T^* \ : (s \overset{w}{\Rightarrow} s') \Rightarrow$ *(a) ou (b) ou (c) avec*
(a) $\exists E \in \Pi_{T_m}, s \overset{E}{\Rightarrow} s'$ *avec* $[w] = [E]$
(b) $\exists F \in \Pi_{T_m}, \exists E \subset F \ : s \overset{E.w_1}{\Rightarrow} s'$ *avec* $[w] = [\mathcal{E}.w_1]$ *et* $F \setminus E \wr w_1$ *où* $\mathcal{E} \in Seq(E)$
(c) $\exists t \in T_u, s \overset{t.w_1}{\Rightarrow} s'$ *avec* $[w] = [t.w_1]$

Démonstration Démonstration du lemme de normalisation
Soit $s \in S$ et $w, w_1 \in T^*$ tels que $s \xrightarrow{w} s'$ et $\pi(s, \emptyset, \epsilon, w) = (E, w_1)$
D'après la construction de la fonction π, trois cas se produisent :
- $E = \{t\}$ et $t \in T_U$
- $E \in \Pi_{T_M}$
- $E \notin \Pi_{T_M}$

Les deux premiers cas sont similaires et le résultat découle de la construction de π.

Détaillons le troisième cas : Par construction, $E \in Step(T, \#^C, s)$, la condition CB_2 implique qu'il existe $F \in \Pi_{T_m}$, vérifiant $E \subseteq F$; La propriété (1) de π implique l'égalité des traces et la propriété (2) que $(F \setminus E) \wr ||w_1||$.

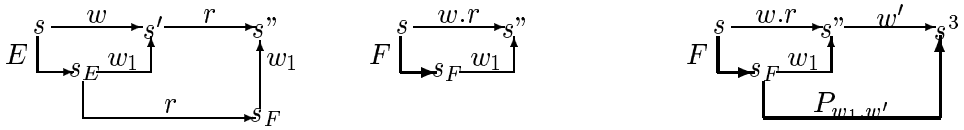
4.3 Démonstration de la condition (4) de la définition des graphes de pas couvrants

Muni du lemme de normalisation, la condition (4) est prouvée par une simple récurrence sur $|\omega|$. Ceci est évident lorsque $|\omega| = 0$.
La propriété (4) des pas couvrants est supposée vraie jusqu'au rang k. Soit ω tel que $|\omega| = k+1$. Le lemme de normalisation s'applique et trois cas se produisent :
Cas (a) : $\exists E \in \Pi_{T_m}, s \stackrel{E}{\Rightarrow} s'$ avec $[w] = [E]$, donc la réponse est directe est triviale.
Cas (b) : $\exists F \in \Pi_{T_m}, \exists E \subset F \ : s \stackrel{E.w_1}{\Rightarrow} s'$ avec $[w] = [\mathcal{E}.w_1]$ et $R \wr w_1$ où $\mathcal{E} \in Seq(E)$ et $E \cup R = F$. Soit l'état s_E tel que $s \stackrel{E}{\Rightarrow} s_E$.
Donc $s_E \stackrel{R}{\twoheadrightarrow}$ et $s_E \stackrel{w_1}{\longrightarrow} s'$ avec $R \wr w_1$. Par conséquent on peut appliquer la propriété du losange qui donne : $\forall r \in Seq(R)$, $s_E \stackrel{r.w_1}{\longrightarrow} s"$ et $s_E \stackrel{w_1.r}{\longrightarrow} s"$ (cf Figure III.8.a)

Donc on est ramené à la situation décrite par la Figure III.8.b car $E \cup R = F$, avec $[w.r] = [F.w_1]$.

L'hypothèse de récurrence peut être utilisée à partir de l'état s_F ; en effet on a $s_F \stackrel{w_1}{\longrightarrow} s"$ avec $|w_1| \leq k$. Donc $\exists w' \in T^*, \exists P_{w_1.w'} \in \Pi^*_{T_M} \ : s_F \stackrel{w_1.w'}{\longrightarrow} s^3, s_F \stackrel{P_{w_1.w'}}{\twoheadrightarrow} s^3$ et $[w_1.w'] = [P_{w_1.w'}]$ (cf Figure III.8.c).

Finalement, $s \stackrel{w.r.w'}{\longrightarrow} s^3$ et $s \stackrel{F.P_{w_1.w'}}{\twoheadrightarrow} s^3$. Or comme $[w.r] = [F.w_1]$ et $[w_1.w'] = [P_{w_1.w'}]$, on a le résultat : $[w.r.w'] = [F.P_{w_1.w'}]$.



III.8.a III.8.b III.8.c

Figure III.8: Cas (b)

Cas (c) : $\exists t \in T_u, s \stackrel{t.w_1}{\Rightarrow} s'$ avec $[w] = [t.w_1]$ donc $s \stackrel{\{t\}}{\rightarrow}_R s_1$ et $s_1 \stackrel{w_1}{\Rightarrow} s'$. L'hypothèse de récurrence s'applique alors sur s_1.

5 Graphe des pas de conflits croisés

Une façon simple de respecter la condition CB_2 est de ne considérer que des pas de transitions maximaux. Comme l'on dispose d'une relation de conflit qui est une relation d'équivalence ($\#$), les ensembles de conflits associés forment une partition de l'ensemble des transitions et l'ensemble des pas maximaux peut être obtenu en considérant l'orthoproduit [PF90] de l'ensemble des classes d'équivalence de la relation $\#$

Définition 53 *Orthoproduit*
Soit un ensemble d'ensembles $\mathbb{E} = \{E_1, E_2, \ldots E_n\}$. *On définit l'orthoproduit de* $\mathbb{E}$ *par :*

$$\Pi_C(\mathbb{E}) = \{\{e_1, e_2, \ldots e_n\} \ : (e_1, e_2, \ldots e_n) \in E_1 \times E_2 \ldots \times E_n\}$$

Définition 54 *Pas de conflits croisés*
On considère $\Pi_C(T_m/\#)$, *l'ensemble des pas dits de conflits croisés.*

Propriété 15
$-\ \pi \in \Pi_C(T_m/\#\!\!\#) \Rightarrow \pi \in Step(T_m, \#\!\!\#^C)$ *et a fortiori* $\pi \in Step(T_m, \wr)$
$-\ \pi \in \Pi_C(T_m/\#\!\!\#), t \notin \pi \Rightarrow \pi \cup \{t\} \notin Step(T_m, \#\!\!\#^C)$

Ces propriétés sont les *conséquences directes de la définition de l'orthoproduit* Π_C *est du fait que* $\#\!\!\#^C \subset \wr$

On considère l'algorithme dérivé du schéma général (cf table III.1) en considérant les fonctions définies dans la Table III.3.

$T_M(E, \wr) = \{t \in E \mid t'[\#]t \Rightarrow t' \in E\}$
$T_U(E, \wr) = E \setminus T_M(E, \wr)$
$\Pi(E, \#\!\!\#) = \Pi_C(E/\#\!\!\#)$

Table III.3: Définitions des fonctions pour la construction d'un *GPC*

Propriété 16 *L'algorithme de la Table III.1 avec les fonctions de la Table III.3 construit un graphe de pas couvrants.*

Démonstration Les conditions CA_1, CA_2 et CB_1 sont trivialement satisfaites, CB_2 est une conséquence des propriétés des pas de conflits croisés.

Exemple 2 *Les exemples de graphe de pas couvrants présentés précédemment sont obtenus par cet algorithme. Dans le cas du scheduler de Milner, présenté en 1, le gain est exponentiel : pour n sites, le STE classique compte* $n \times 2^n$ *états et* $(n^2 + n) \times 2^{n-1}$ *transitions alors que le GPC engendré compte* $n + 1$ *états et* $n + 1$ *transitions.*

6 Exemple de construction d'un graphe de pas couvrants

L'objectif de cette section est de donner, dans le détail, chaque étape de la construction d'un graphe de pas couvrants sur un exemple particulier. Nous allons considérer un exemple un peu différent de l'exemple II.11 car sur cet exemple la fermeture transitive de la relation de conflit est identique à la relation de conflit elle même. Nous considérons donc le réseau de Petri de la Figure III.9 qui présente du parallélisme, du conflit et le cas de confusion.

On considère la relation d'indépendance définie dans la présentation des ordres partiels par $t_1 \wr t_2$ ssi ${}^\bullet t_1 \cap {}^\bullet t_2 = \emptyset$. La relation de conflit noté $\#$ est la relation complémentaire.

Sur le réseau de Petri considéré, on obtient donc les conflits suivants : $t_0 \# t_1$, $t_3 \# t_0$ et $t_5 \# t_6$. En calculant la fermeture transitive de la relation de conflit on obtient $t_3 \#\!\!\# t_1$ en plus de $t_0 \#\!\!\# t_1$, $t_3 \#\!\!\# t_0$ et $t_5 \#\!\!\# t_6$. Cette analyse structurelle du réseau de Petri permet le calcul d'un partitionnement de l'ensemble des transitions du réseau de Petri. Ce partitionnement est statique : la relation de conflit est définie par rapport à la structure du réseau de Petri, par conséquent le partitionnement est calculé une seule fois avant le début de l'exploration et non pas dans chaque état exploré. Pour notre exemple le partitionnement de l'ensemble des transitions est représenté figure III.10.

D'après la définition d'un pas (ensemble de transitions indépendantes), tout sous-ensemble π qui n'a pas deux transitions de la même partition est un pas. On obtient donc les pas suivants : $\{t_0\}$, $\{t_0; t_2\}$, $\{t_0; t_2; t_5\}$, $\{t_1; t_2; t_4; t_6\}\ldots$

Il faut noter qu'avec la définition précédente, l'ensemble $\{t_1, t_2\}$ est un pas, pourtant ces deux transitions ne seront jamais simultanément sensibilisées dans ce modèle, par conséquent ce pas ne sera jamais sensibilisée.

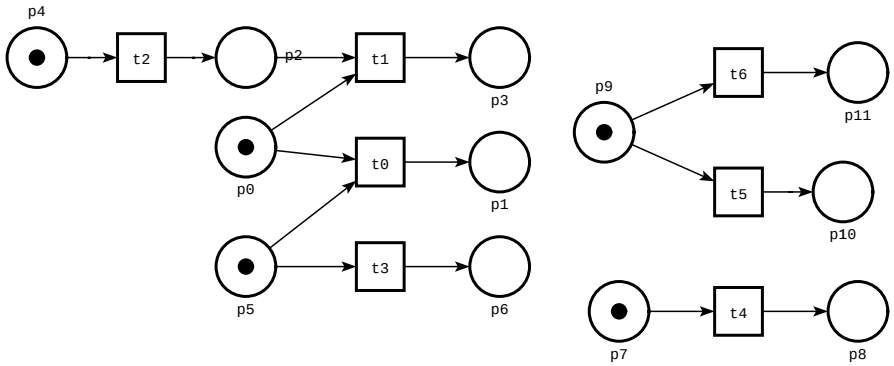


Figure III.9: Réseau de Petri servant d'exemple

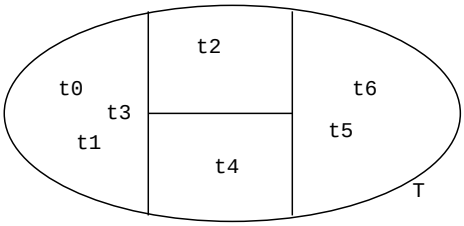


Figure III.10: Partitionnement des transitions par la fermeture transitive de la relation de conflit

L'exploration peut commencer depuis l'état initial : dans l'état initial 6 transitions sont sensibilisées : $t_0, t_2, t_3, t_4, t_5, t_6$. Puisque les transitions t_3 et t_0 sont en conflit avec la transition t_1 non sensibilisée, elles sont mises dans l'ensemble T_u. Donc on obtient $T_m = \{t_2, t_4, t_5, t_6\}$ et $T_u = \{t_0, t_3\}$. En considérant d'une part l'ensemble T_m et d'autre part le partitionnement préliminaire, on calcule l'orthoproduit de $\{\{t_2\}; \{t_4\}; \{t_5, t_6\}\}$, qui nous donne deux pas possibles : $\{t_2, t_4, t_5\}$ et $\{t_2, t_4, t_6\}$. L'exploration de ces deux pas depuis l'état initial nous amène dans deux nouveaux états 11 et 9. Les transitions de l'ensemble T_u sont quand à elles, explorées isolément pour donner les états 1 et 6 dans le graphe de la figure III.11. Dans chacun de ces nouveaux états, on considère l'ensemble des transitions sensibilisées pour calculer les ensembles T_m et T_u.

Sur cet exemple, on observe qu'il ne suffit pas de remplacer chaque pas par l'hypercube composé des entrelacements possibles des transitions qu'il contient pour retrouver le graphe complet. Par exemple la séquence de transitions $t_2.t_3.t_5.t_1.t_4$ est une séquence tirable dans le réseau de Petri. La propriété de couverture des graphes de pas couvrants, garantit l'existence d'une séquence de pas dans le graphe de pas couvrants, telle que les deux séquences (en prolongeant éventuellement la séquence de transitions) soient équivalentes au sens de Mazurkiewicz. En effet les séquences de pas $\{t_2, t_4, t_5\}.\{t_3\}.\{t_1\}$, $\{t_2, t_4, t_5\}.\{t_1\}.\{t_3\}$ et également $\{t_3\}.\{t_2, t_4, t_5\}.\{t_1\}$ couvrent la séquence et sont équivalentes.

Si on considère le graphe construit en remplaçant chaque « pas » par l'hypercube formé sur les transitions du « pas », ce graphe ne contient pas la séquence de transitions $t_2.t_3.t_5.t_1.t_4$. En fait le graphe obtenu est un « automate trace » : toute séquence d'exécution a une séquence dans ce graphe qui lui soit équivalente (au sens de Mazurkiewicz).

Pour visualiser la réduction obtenue sur cet exemple, le graphe des marquages exhaustif est donné figure III.12. Celui-ci comporte 48 états et 116 transitions.

La section suivante montre un exemple industriel de l'utilisation des GPC que nous avons

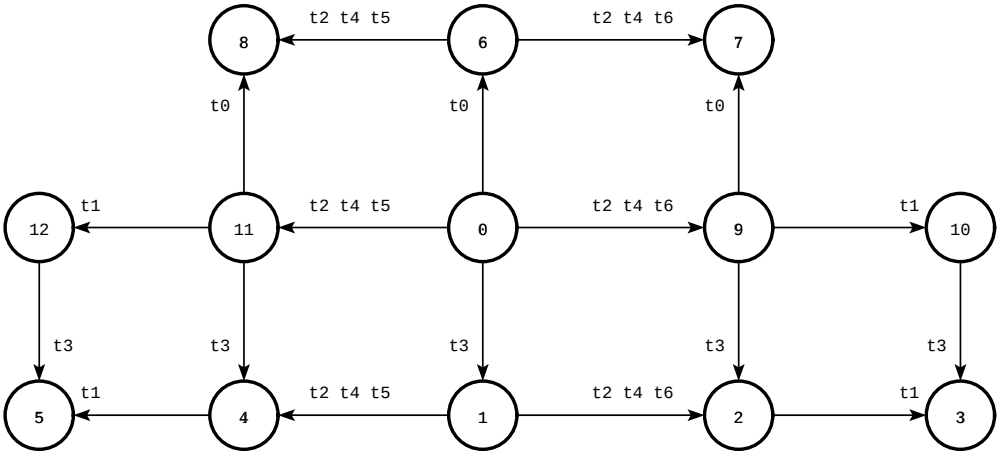


Figure III.11: Graphe de pas couvrants

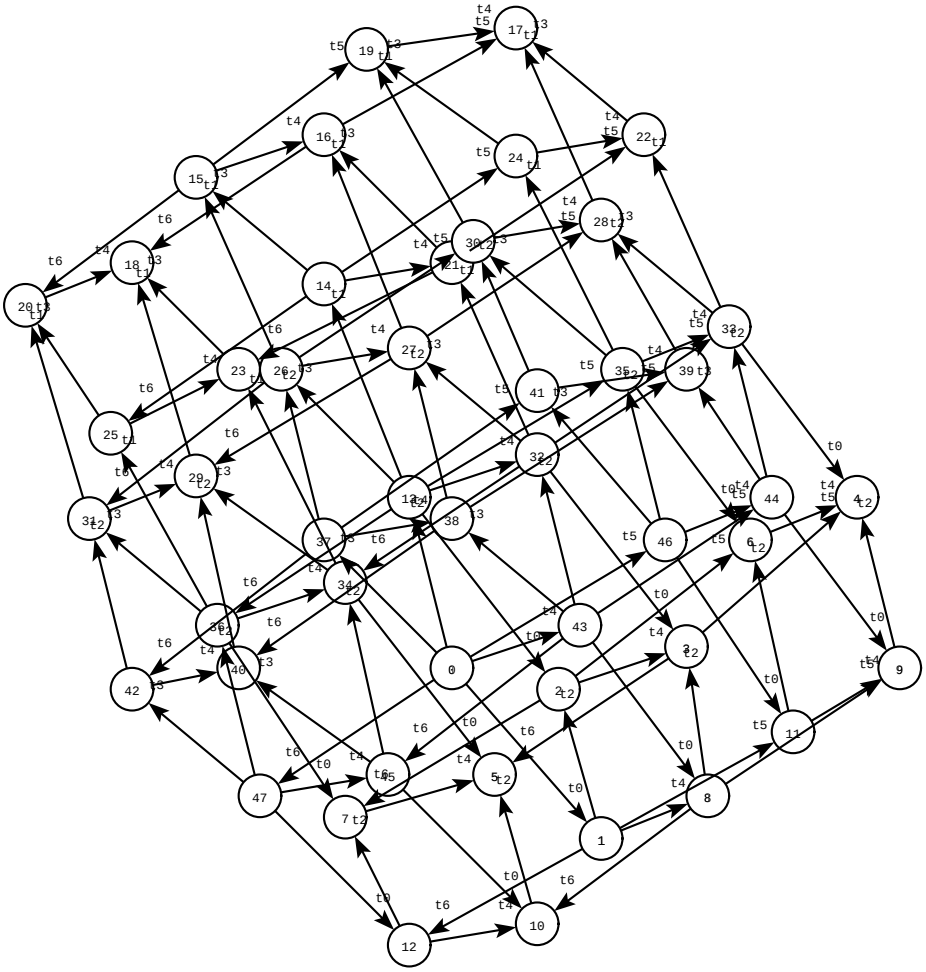


Figure III.12: Graphe exhaustif

implémenté dans l'outil *Tina*.

7 Expérimentation sur un exemple industriel

Un exemple industriel de taille conséquente a été proposé par Hubert Garavel sur la mailing liste internationale « Petri Nets ». L'objectif est de vérifier la quasi-vivacité du système (absence de transitions mortes).

Le modèle décrit en réseau de Petri est constitué de 485 places et 776 transitions, il a été obtenu à partir d'une étude de cas industrielle réelle. Le réseau de Petri a été généré automatiquement à partir d'une spécification LOTOS (8 500 lignes de code LOTOS + 3 000 lignes de code C) en utilisant le compilateur CAESAR de CADP (http ://www.inrialpes.fr/vasy/cadp). Une construction brutale du graphe des marquages échoue (plus de 109 millions d'états), par contre il a de bonnes propriétés structurelles dues au fait que le réseau de Petri a été généré à partir d'une spécification en algèbre de processus. En particulier :

- il est 1-sauf (au plus un jeton par place).
- l'ensemble des places est partitionné en « composants séquentiels ». Dans chaque composant séquentiel il y a au plus une place qui a un jeton.
- il y a 60 composants séquentiels, par conséquent chaque marquage a au plus 60 jetons.

Cet exemple industriel de taille conséquente permet de comparer différentes techniques et plusieurs outils utilisés pour résoudre ce problème. Les différentes conclusions obtenues avec les différents logiciels semblent converger : ce système comporte environ 9.8×10^{21} marquages accessibles et est quasi-vivant.

Nous avons testé cet exemple avec l'outil *Tina* sur un station SunFire V880 900MHz. Pour cela nous avons utilisé la construction des *GPC* puisque elle préserve la quasi-vivacité (puisque toutes les transitions sensibilisées sont tirées). De plus une réduction du réseau de Petri par des règles adaptées des ensembles de Berthelot et Esparza [Ber85, ES01] (préservant la quasi-vivacité) a été ajoutée à *Tina*.

Par ailleurs la vérification de la quasi-vivacité peut-être faite au vol : on maintient à jour au fur et à mesure la liste des transitions déjà rencontrées. Dès que toutes les transitions ont été trouvées tirables la construction s'arrête. Finalement une stratégie qui explore prioritairement les transitions qui n'ont jamais été rencontrées depuis le début a été utilisée.

Différentes stratégies ont alors été essayées :

- Une construction directe en largeur d'abord avec les *GPC* (sans les réductions du réseau de Petri). Un *GPC* partiel est obtenu en 7 minutes avec 327 052 marquages et 1 110 112 transitions. Puisque ce graphe contient toutes les transitions, la propriété de quasi-vivacité du système est vérifiée.
- Une réduction du réseau de Petri, suivie par une construction *GPC* en profondeur d'abord. La réduction permet de remplacer le réseau de Petri initial (485 places, 776 transitions) par un plus petit (133 places, 185 transitions). On obtient alors un *GPC* partiel avec 644 519 marquages et 1 527 037 transitions en 1 heure et 32 minutes. Toutes les transitions du réseau de Petri réduit sont trouvées tirables. Comme la réduction préserve la propriété de quasi-vivacité, aucune transition du réseau de Petri initial n'est morte.
- Une réduction du réseau de Petri, suivie par une construction *GPC* en largeur d'abord. On arrive alors à un *GPC* partiel avec 41 568 marquages et 106 058 transitions, calculé en 6 secondes. Aucune transition n'est trouvée morte.

Cet exemple a permis de comparer les performances de *Tina* et des techniques utilisées avec les outils existants. A notre connaissance seulement quatre logiciels ont réussi à vérifier cette propriété de vivacité sur cet exemple :

- *Smart* testé par Gianfranco Ciardo et Radu Siminiceanu, est basé sur une technique de génération d'états symboliques et une technique de saturation pour ordonner les variables,
- *LoLA* par Karsten Schmidt de Berlin, utilise également les ordres partiels,
- *Versify* de Oriol Roig utilise une technique de construction d'états symboliques avec des BDD,
- et *Tina*.

Chapter IV

Graphe de pas préservant les formules bégayement invariantes

La construction de pas, dans les graphes de pas couvrants, supprime l'entrelacement de certaines actions. Par conséquent des formules LTL qui peuvent exprimer par exemple que telle propriété (exprimée par une combinaison des marquages) est vraie jusque à ce que telle autre devienne vraie, ne peuvent pas être vérifiées sur un graphe de pas couvrants classique. Nous avons donc défini une variante appelée $LGPC$ qui consiste à ne construire que les pas qui ne sont pas « gênants » par rapport à la propriété à laquelle on s'intéresse. Nous allons donner une définition dénotationnelle des $LGPC$ (sans nous soucier de la façon opérationnelle de construire un tel graphe). Nous démontrons alors que les formules bégayement invariantes sont préservées par les $LGPC$. Enfin nous donnons un algorithme particulier permettant de construire un $LGPC$, avec la démonstration que cet algorithme construit bien un $LGPC$.

1 Introduction

Propriété 17 *Les GPC ne préservent pas les formules de LTL*

Nous allons considérer l'exemple du scheduler de Milner que nous allons modifier légèrement en ajoutant une simple boucle indépendante comme sur la Figure IV.1. L'intérêt de cette boucle est d'introduire de la divergence dans le modèle de base. Le système peut exécuter indéfiniment la transition t_0 depuis n'importe quel marquage accessible.

Sur ce nouveau modèle on s'intéresse à la propriété « M_1 arrive une infinité de fois » qui s'écrit en LTL par $\varphi = [] <> M_1$ (Toujours inévitablement M_1 arrive). Cette formule est fausse dans le modèle de la Figure IV.1 ; En effet la séquence infinie $\xrightarrow{A_1}\xrightarrow{t_0}\xrightarrow{t_0} \ldots$ ne satisfait pas la formule φ. Après le tir de la transition A_1 la place M_1 n'est pas marquée et elle reste indéfiniment non marquée par cette séquence.

Si on construit le graphe de pas couvrants avec ce modèle, on obtient le graphe de la figure IV.2 : Ce système de transitions étiquetées est semblable à celui que l'on obtient pour le scheduler classique. La transition t_0 est indépendante de toutes les autres transitions, elle est donc ajoutée à tous les pas. Par conséquent le nombre d'états du système reste exactement le même. Les valuations (ou marquages correspondants) des états de ce graphe sont donnés dans la Figure IV.2.

Toute séquence de ce graphe admet donc une infinité de fois un état avec la valuation M_1. Par conséquent la formule φ ci-dessus est vraie sur le graphe GPC de la Figure IV.2 alors qu'elle est fausse pour le système de la figure IV.1. Ce contre exemple démontre la propriété 17.

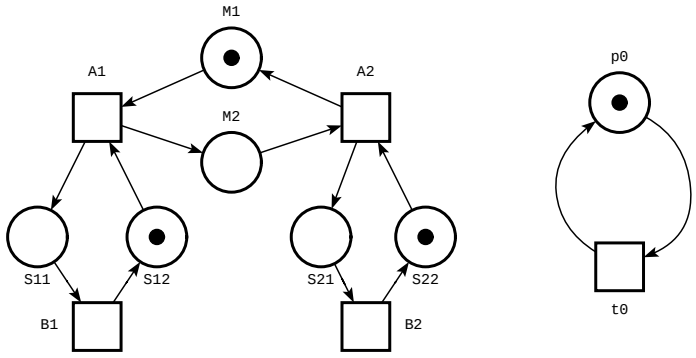
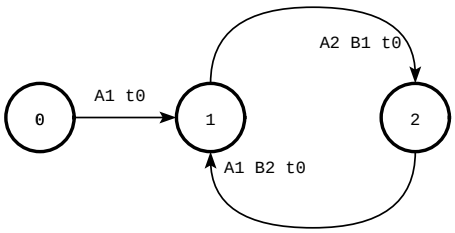


Figure IV.1: Scheduler de Milner avec une boucle supplémentaire indépendante



État	Marquage
0	$M_1\ S_{12}\ S_{22}\ P_0$
1	$M_2\ S_{11}\ S_{22}\ P_0$
2	$M_1\ S_{12}\ S_{21}\ P_0$

Figure IV.2: Graphe de pas couvrants pour le scheduler de Milner modifié

L'objectif de ce chapitre est donc de définir une nouvelle technique de graphe de pas préservant les formules logiques de LTL.

2 Construction d'un structure bégayement équivalente

2.1 Quelles formules préserver ?

Vouloir construire un graphe qui préserve toutes les formules de la logique LTL conduirait à ne faire aucune réduction car le pouvoir d'expression de la logique LTL est très important. Inversement on pourrait vouloir construire un graphe spécifiquement par rapport à la formule que l'on souhaite vérifier afin de construire un graphe réduit de taille minimale, mais outre le fait que nous ne savons pas construire ce graphe minimal par rapport à une formule, cette approche obligerait à reconstruire un graphe pour chaque formule que l'on souhaite vérifier.

La démarche que nous présentons ici se situe entre ces deux extrêmes. Pour cela, nous allons faire deux restrictions nous permettant de réduire la taille du graphe construit.

On peut d'abord remarquer que dans la pratique, les formules « utiles » n'utilisent généralement pas la modalité $Next\ \ \mathcal{X}$; en effet, il n'est pas très important de savoir que tel état sera atteint dans une ou deux étapes, par contre il est important de savoir que l'on aura tel état après tel autre. Cette remarque nous permet de dire que dans la plupart des cas préserver les formules de LTL_{-X} sera suffisant. Pour un coût identique, nous allons préserver une classe de propriétés (légèrement) plus grande qui correspond aux formules qui sont bégayement invariantes. Intuitivement une propriété qui est bégayement invariante est une propriété qui ne différencie pas deux exécutions qui diffèrent seulement par la répétition de certains états. La figure IV.3 représente le pouvoir d'expression de ces différentes classes. Dans la suite nous allons définir formellement

le bégayement et les formules bégayement invariantes.

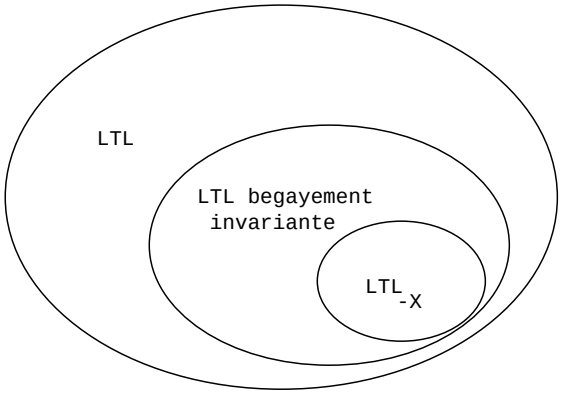


Figure IV.3: Pouvoir d'expression des logiques LTL

La seconde remarque concerne les variables propositionnelles utilisées dans une formule. Une formule LTL est souvent définie sur un sous-ensemble des variables propositionnelles. Nous n'allons donc pas préserver toutes les formules définies sur AP (notées $LTL(AP)$) mais considérer un sous-ensemble AP_{Obs} dit *observable* des propositions atomiques et conserver toutes les propriétés définies sur ce sous-ensemble ; soit $LTL(AP_{Obs})$.

Ces deux remarques nous conduisent à construire un graphe qui préserve les propriétés invariantes par bégayement construites sur un sous-ensemble de variables propositionnelles.

2.2 De LTL à LTL_{-X} en passant par le bégayement

Définition 55 *Séquences bégayement équivalentes (Stuttering equivalent)*

Deux séquences d'états σ, σ' *telles que* $s_0 \longrightarrow s_1 \longrightarrow s_2 \ldots$ *et* $s_0 \longrightarrow s'_1 \longrightarrow s'_2 \ldots$ *sont bégayement équivalentes ssi il existe deux séquences infinies d'entiers* $0 = i_0 < i_1 < \ldots$ *et* $0 = j_0 < j_1 < \ldots$ *telles que* $\forall k \geq 0, \nu(s_{i_k}) = \nu(s_{i_k+1}) = \ldots = \nu(s_{i_{k+1}-1}) = \nu(s'_{j_k}) = \nu(s'_{j_k+1}) = \ldots = \nu(s'_{j_{k+1}-1})$

On note alors $\sigma \sim_{st} \sigma'$.

On rappelle que la fonction ν est la fonction valuation. La Figure IV.4 donne un exemple de deux séquences bégayement équivalentes, en effet on peut définir :

$$i_0 = 0, i_1 = 1, i_2 = 4, i_3 = 5, i_4 = 6 \ldots$$

$$j_0 = 0, j_1 = 2, j_2 = 3, j_3 = 4, j_4 = 5 \ldots$$

Les pointillés sur la figure représentent les séparations faites par ces indices. Les états adjacents possédant le même étiquetage par ν sont appelées des blocs. Remarquons que bien que nos séquences restent indéfiniment avec la valuation $(\neg p, \neg q)$ (à partir de l'état s_4 pour la séquence σ et de l'état s'_3 pour la séquence σ'), les séquences infinies d'indices sont obtenues en faisant une infinité de blocs de taille 1 à partir des indices i_2 et j_2.

En utilisant le bégayement d'une séquence, on peut définir les formules insensibles au bégayement.

Définition 56 *Formule invariante par bégayement*

Une formule φ *de* LTL *est invariante par bégayement ssi pour toutes séquences d'exécution* σ *et* σ' *telles que* $\sigma \sim_{st} \sigma'$ *on a :*

$$\sigma \models \varphi \Leftrightarrow \sigma' \models \varphi$$

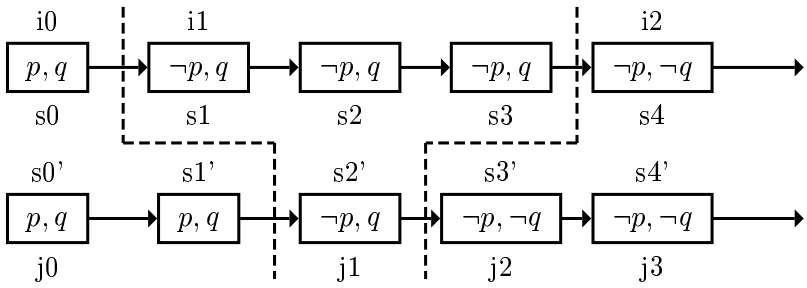


Figure IV.4: Deux séquences bégayement équivalentes

De façon intuitive, une formule invariante par bégayement fait abstraction du nombre d'états adjacents dans lequel une propriété est vraie, mais considère la succession ou l'ordre des différentes valuations dans les états de la séquence. Une formule invariante par bégayement est aussi dite « **fermée par bégayement** » (closed under stuttering).

Déterminer dans le cas général si une formule est bégayement invariante est complexe. La solution de Peled basée sur la construction des tableaux des ω-automates se ramène à un problème PSPACE-complet[PCB98].

Définition 57 *Logique LTL_{-X}*

Les formules LTL_{-X} sont des formules LTL qui n'utilisent pas la modalité $\mathcal{X}$.

LTL_{-X} est définie comme la logique LTL sans l'opérateur $\mathcal{X}$. Cette logique est particulièrement intéressante, car elle constitue un sous-ensemble de la logique des formules fermées par bégayement. Elle permet donc d'avoir des formules dont la fermeture par bégayement est trivialement caractérisable.

Théorème 2 *[CGP00] Toute formule Φ de LTL_{-X} est invariante par bégayement.*

La simplicité de ce théorème par rapport à la complexité de prouver qu'une formule est bégayement invariante explique l'intérêt porté aux propriétés de LTL_{-X}. Cependant certaines propriétés d'équité ne peuvent pas être exprimées en LTL_{-X} alors qu'elles peuvent s'exprimer simplement par des formules bégayement invariantes, il est donc intéressant dans ce cas de montrer que la propriété est invariante par bégayement et qu'elle est donc préservée par les méthodes de réduction proposées. [PCB98] propose de montrer qu'une formule est bégayement invariante par des règles de construction sur les formules. Par exemple les formules $(\neg a \wedge \mathcal{X}a)$ et $(a \wedge \mathcal{X}\neg a)$ permettent de caractériser une variable qui passe de faux à vrai ou l'inverse. Si on pose la formule $\varphi = (\neg a \wedge \mathcal{X}a) \vee (a \wedge \mathcal{X}\neg a)$ alors pour toute formule ψ bégayement invariante, la formule $\psi \mathcal{U} \varphi$ est bégayement invariante. Intuitivement cette formule n'exprime pas dans combien d'états la valuation de la variable change, mais exprime le fait que ψ est vraie tant que la proposition atomique a n'a pas changé de statut en passant de vrai à faux ou de faux à vrai ; si on supprime les répétitions d'états qui ont la même valuation le changement aura toujours lieu, donc cette formule est bien bégayement invariante. Cet exemple illustre une formule LTL utilisant l'opérateur $\mathcal{X}$ qui est bégayement invariante : il est donc important de noter que la technique présentée ici ne se limite pas aux formules de LTL_{-X}, mais peut être utilisée pour toutes les formules bégayement invariantes.

2.3 Structure bégayement équivalente

Notre objectif est de construire un graphe réduit qui permette de vérifier une formule bégayement invariante : un tel graphe est dit bégayement équivalent au graphe exhaustif. Pour cela définissons d'abord une notion d'inclusion.

Définition 58 *Structures bégayement incluses*
Une structure de Kripke $\mathcal{SK}$ est bégayement incluse dans une structure de Kripke $\mathcal{SK}'$ si et seulement si

1. *$\mathcal{SK}$ et $\mathcal{SK}'$ ont le même état initial s_0.*
2. *Pour toute séquence d'exécution σ partant de l'état initial de $\mathcal{SK}$, il existe une séquence σ' partant de l'état initial de $\mathcal{SK}'$ telle que $\sigma \sim_{st} \sigma'$.*

On note alors $\mathcal{SK} \subseteq_{st} \mathcal{SK}'$.

Définition 59 *Structures bégayement équivalentes*
Deux structures de Kripke $\mathcal{SK}$ et $\mathcal{SK}'$ sont bégayement équivalentes ssi

$$\mathcal{SK} \subseteq_{st} \mathcal{SK}' \text{ et } \mathcal{SK}' \subseteq_{st} \mathcal{SK}$$

Cette définition exprime le fait que pour toute séquence de transitions de $\mathcal{SK}$ il existe une séquence bégayement équivalente dans $\mathcal{SK}'$ et réciproquement.

Pour pouvoir vérifier une formule bégayement invariante sur un graphe réduit il nous reste à voir comment construire un graphe bégayement équivalent au graphe complet. Pour cela nous avons besoin de voir le lien entre les transitions du système et le bégayement.

2.4 Transitions et états observables

Comme les explorations « ordre partiel » opèrent sur les transitions et non sur les propositions d'état, il faut être capable d'associer à chaque sous-ensemble AP_{Obs} un sous-ensemble T_{Obs} de transitions qui lui soit compatible. Pour faciliter cette association nous requérons que le double étiquetage (des états et des transitions) existant dans les SKE soit homogène.

Définition 60 *SKE Homogène*
Soit $\mathcal{SKE} = \langle AP, T, S, s_0, \{\overset{t}{\longrightarrow}\}_{t \in T}, \nu \rangle$. $\mathcal{SKE}$ est homogène ssi
$\forall P \subseteq AP, \forall s, s_1, s', s'_1 \in S, \forall t \in T$:

$$Si\ s \overset{t}{\longrightarrow} s' \text{ et } s_1 \overset{t}{\longrightarrow} s'_1 \text{ alors } \nu_P(s) = \nu_P(s_1) \Rightarrow \nu_P(s') = \nu_P(s'_1)$$

où $\nu_P : S \mapsto AP$ est définie par $\nu_P(s) = \nu(s) \cap P$

Intuitivement cela signifie qu'une transition donnée a la même conséquence en terme de valuation qu'elle soit appliquée dans un état ou dans un autre.

Dans la pratique la restriction à des structures de Kripke étiquetées homogènes n'est pas une grande contrainte. En particulier toute SKE obtenue à partir d'un produit synchronisé d'automates ou obtenue à partir d'un graphe des marquages d'un réseau de Petri est par construction homogène.

Une formule de LTL_{-X} est définie sur un sous-ensemble des variables propositionnelles AP, ce qui signifie que seules ces variables propositionnelles importent pour l'évaluation de la formule. Cet ensemble dit observable est noté AP_{Obs}.

Par conséquent à un sous-ensemble de variables atomiques AP_{Obs}, on peut associer un sous-ensemble de transitions dites observables T_{Obs}.

Définition 61 *Transition Observable par rapport à AP_{Obs} [Pel98]*

Soit $\mathcal{SKE} = \langle AP, T, S, s_0, \{\overset{t}{\longrightarrow}\}_{t\in T}, \nu\rangle$ une SKE homogène et $AP_{Obs} \subseteq AP$.
Une transition $t \in T$ est dite observable par rapport à AP_{Obs} ssi

$$\exists s_1, s_2 \in S \ : s_1 \overset{t}{\longrightarrow} s_2 \text{ et } \nu_{Obs}(s_1) \neq \nu_{Obs}(s_2)$$

où $\nu_{Obs} : S \mapsto AP$ est définie par $\nu_{Obs}(s) = \nu(s) \cap AP_{Obs}$
L'ensemble des transitions observables sera noté T_{Obs} dans la suite.

Intuitivement une transition observable est une transition qui change la valuation d'une proposition atomique observable, donc son tir peut changer l'évaluation de la formule.

Par conséquent, l'ordre dans lequel sont exécutées les transitions observables est important pour vérifier la satisfaction de la formule. Nous allons donc définir des fonctions permettant de récupérer la partie à observer d'une séquence de transitions ou d'une séquence de pas.

Définition 62 *Morphisme cachant*

Soit T un ensemble de transitions et $T_{Obs} \subseteq T$ le sous-ensemble des transitions observables. ϵ décrit une séquence vide de T^*.

On définit $f_{Obs} : T^\omega \mapsto T^\omega_{Obs}$ le morphisme cachant par :

$$f_{Obs}(\epsilon) = \epsilon$$
$$\text{et } \forall t \in T, w \in T^\omega, f_{Obs}(t.w) = \begin{cases} t.f_{Obs}(w) \text{ si } t \in T_{Obs} \\ f_{Obs}(w) \text{ sinon} \end{cases}$$

Le morphisme cachant supprime d'une séquence de transitions finie ou infinie, toutes les transitions qui ne sont pas observables.

Ces définitions nous permettent d'écrire le lemme suivant qui exprime le fait que deux séquences, n'ayant pas obligatoirement la même longueur, ayant pour origine des états qui ont la même valuation par rapport à AP_{Obs}, et la même séquence de transitions observables, sont bégayement équivalentes. Ce lemme sera utilisé pour démontrer que deux graphes sont bégayement équivalents.

Lemme 2 *Soient $w = t_1.t_2\ldots$ et $w' = t'_1.t'_2\ldots$ deux séquences d'exécution, ayant respectivement pour origine s_0 et s'_0 vérifiant $\nu_{Obs}(s_0) = \nu_{Obs}(s'_0)$ et $\sigma = s_0 s_1 \ldots$, $\sigma' = s'_0 s'_1 \ldots$ telles que $s_0 \overset{t_1}{\longrightarrow} s_1 \overset{t_2}{\longrightarrow} s_2 \ldots$ et $s'_0 \overset{t'_1}{\longrightarrow} s'_1 \overset{t'_2}{\longrightarrow} s'_2 \ldots$, alors*

$$f_{Obs}(w) = f_{Obs}(w') \Rightarrow \sigma \sim_{st} \sigma'$$

Démonstration Soient $w = t_1.t_2\ldots$ et $w' = t'_1.t'_2\ldots$ deux séquences telles que $f_{Obs}(w) = f_{Obs}(w')$, avec $\sigma = s_0 s_1 \ldots$, $\sigma' = s'_0 s'_1 \ldots$ tels que $s_0 \overset{t_1}{\longrightarrow} s_1 \overset{t_2}{\longrightarrow} s_2 \ldots$ et $s'_0 \overset{t'_1}{\longrightarrow} s'_1 \overset{t'_2}{\longrightarrow} s'_2 \ldots$

Soit (i_k) la suite strictement croissante des indices des transitions observables de w. Donc $f_{Obs}(w) = f_{Obs}(t_1.t_2\ldots) = t_{i_1}.t_{i_2}\ldots$
Si la séquence $f_{Obs}(w)$ est finie, soit t_l la dernière transition observable de w, alors on pose $\forall k \geq l, i_{k+1} = 1 + i_k$.

Donc d'après cette construction, $\forall n \geq 0, \forall k \in]i_n; i_{n+1}[\quad t_k \notin T_{Obs}$ Donc $\forall n \geq 0, \forall k, l \in [i_n; i_{n+1}[,\ \nu(s_k) = \nu(s_l)$.

On définit exactement de la même manière la suite (j_k) par rapport aux transitions observables de w'.

De plus $f_{Obs}(w) = f_{Obs}(w') \Rightarrow \forall k \geq 1, t_{i_k} = t'_{j_k}$. Donc en utilisant la définition 60 d'un SKE homogène et $\nu_{Obs}(s_0) = \nu_{Obs}(s'_0)$ on démontre facilement par récurrence que $\forall n \geq 1, \forall k \in [i_n; i_{n+1}[,\ k' \in [j_n; j_{n+1}[,\ \nu(s_k) = \nu(s'_{k'})$. En prenant $i_0 = j_0 = 0$ nous avons bien défini deux suites strictement croissantes (i_k), (j_k) d'indices de débuts de blocs (définition 55) des séquences d'états σ et σ' donc $\sigma \sim_{st} \sigma'$.

Remarquons que l'implication inverse est fausse : soient deux transitions t_i et t'_i différentes et observables telles que $s_{i-1} \xrightarrow{t_i} s_i$ et $s'_{i-1} \xrightarrow{t'_i} s'_i$ et $\nu(s_{i-1}) = \nu(s'_{i-1})$ et $\nu(s_i) = \nu(s'_i)$, donc on aura $s_{i-1}.s_i \sim_{st} s'_{i-1}.s'_i$ et pourtant $f_{Obs}(t_i) \neq f_{Obs}(t'_i)$. Donc deux séquence bégayement équivalentes, n'ont pas forcément la même séquence de transitions observables.

Le Lemme 2 est très important puisqu'il nous permet à partir de la définition de structures bégayement incluses (Définition 58) de définir un nouveau lemme permettant de démontrer l'inclusion.

Lemme 3 *Structures bégayement incluses*
Une structure de Kripke $\mathcal{SK}$ est bégayement incluse dans une structure de Kripke $\mathcal{SK}'$ si

1. *$\mathcal{SK}$ et $\mathcal{SK}'$ ont le même état initial s_0.*
2. *Pour toute séquence d'exécution w partant de l'état initial de $\mathcal{SK}$, il existe une séquence w' partant de l'état initial de $\mathcal{SK}'$ telle que $f_{Obs}(w) = f_{Obs}(w')$*

Remarquons que dans le lemme nous n'avons plus un « si et seulement si », puisque ce sont des conditions suffisantes mais pas nécessaires.

2.5 Conclusion

Cette section nous amène aux conclusions suivantes : il n'est pas judicieux de vouloir préserver LTL dans sa globalité, nous nous intéressons donc aux formules bégayement invariantes. De plus étant donné la formule φ que l'on veut vérifier, on en déduit l'ensemble AP_{Obs} de variables propositionnelles $AP_{Obs} \subseteq AP$ utilisées par la formule. De cet ensemble on peut en déduire un ensemble de transitions observables $T_{Obs} \subseteq T$. La Figure IV.5 illustre l'enchaînement des différents théorèmes et lemmes définis dans cette section. Si le graphe réduit préserve toutes les séquences observables (par rapport à T_{Obs}), alors d'après le Lemme 3 et la Définition 59 il sera bégayement équivalent au graphe complet, ce qui signifie par définition que toutes les formules bégayement invariantes décrites sur AP_{Obs} seront préservées par ce graphe.

En particulier toutes les formules de LTL_{-X} écrites sur AP_{Obs} sont bégayement invariantes, donc elles seront préservées par ce graphe réduit.

3 Définition dénotationnelle des $LGPC$

3.1 Vers un graphe de pas préservant les formules bégayement invariantes....

Nous avons défini la partie observable d'une séquence de transitions, il nous faut maintenant la transposer aux séquences de pas. La règle de base d'un pas, est que toutes les transitions le constituant doivent pouvoir être explorées dans l'ordre que l'on veut. Intuitivement l'ordre des transitions qui ne sont pas observables, n'a pas d'importance, par contre l'ordre des transitions

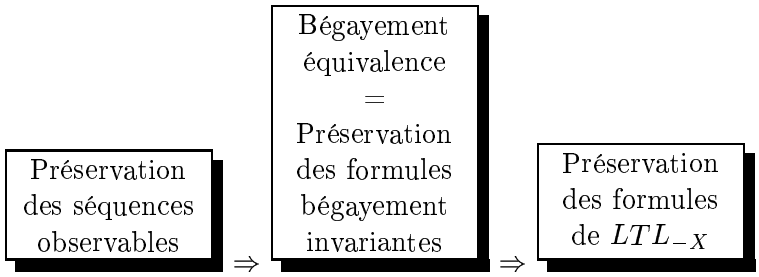


Figure IV.5: Préservation des formules de LTL_{-X}

observables est important. Pour ne pas perdre le séquencement des transitions observables on va interdire d'avoir plusieurs transitions observables dans le même pas de transitions.

Définition 63 *Ensembles compatibles*
Soit Π *une séquence de pas* $\Pi = \pi_1\pi_2\ldots$ *avec* $\forall i, \pi_i \in \mathcal{P}(T)$.
T_{Obs} *et* Π *sont* **compatibles** *ssi* $\forall \pi_i \; Card(\pi_i \cap T_{Obs}) \leq 1$
où $Card(F)$ *donne le nombre d'éléments de l'ensemble* F

On étend donc le morphisme cachant aux séquences de pas.

Définition 64 *Morphisme cachant d'une séquence de pas*
Pour T_{Obs} *et* Π *compatibles, on définit* $I\!F_{Obs}$, *un* **morphisme cachant** *défini comme suit :*
$I\!F_{Obs} : \mathcal{P}(T_R)^\omega \mapsto T_{Obs}^\omega$,
$I\!F_{Obs}(\{\epsilon)\} = \epsilon$
et $I\!F_{Obs}(\pi_1\pi_2\pi_3\ldots) = \begin{cases} t.I\!F_{Obs}(\pi_2\pi_3\ldots) & si\ \pi_1 \cap T_{Obs} = \{t\} \\ I\!F_{Obs}(\pi_2\pi_3\ldots) & sinon \end{cases}$

Lemme 4 *Soient* T_{Obs} *et* Π *compatibles, alors* $I\!F_{Obs}(\Pi) = f_{Obs}(seq(\Pi))$
avec $seq(\Pi)$ *une linéarisation quelconque de la séquence de pas, en séquence de transitions.*

3.2 Graphe de pas bégayement équivalent

Définition 65 *Graphe de pas couvrants LGPC*
Soit $\mathcal{STE} = \langle S, s_0, T, \{\overset{t}{\longrightarrow}\}_{t\in T}\rangle$ *un STE,* $\wr$ *une relation d'indépendance sur* $\mathcal{STE}$ *et* $\# = \wr^C$.
$\mathcal{STE}_R = \langle S_R, s_o, T_R, \longrightarrow_R\rangle$ *est un* **graphe de pas couvrants bégayement équivalent** $LGPC$ *par rapport à* $\wr$ *et* T_{Obs}, *ssi*

1. $\mathcal{STE}_R$ *est un graphe de pas couvrants*
2. $\forall \Pi \in T_R$: Π *et* T_{Obs} *sont compatibles*
3. $\forall s \in S_R : s \overset{\omega}{\Longrightarrow}$ *implique* $\begin{cases} \exists \omega' \in {f_{Obs}}^{-1}(\epsilon), \exists \Pi_{\omega.\omega'} \in {T_R}^\omega : \\ - \; s \overset{\Pi_{\omega.\omega'}}{\rightarrow}_R \\ - \; [\omega.\omega'] = [\Pi_{\omega.\omega'}] \\ - \; f_{Obs}(\omega) = I\!F_{Obs}(\Pi_{\omega.\omega'}) \end{cases}$

La condition (1) assure que tout $LGPC$ est un GPC.
La condition (2) assure qu'un pas de transitions contient au plus une transition observable.
La condition (3) est une extension de la condition de couverture des graphes de pas couvrants. Elle assure que l'on a des séquences de transitions - y compris infinies - observables identiques entre le graphe complet et le graphe réduit.

4 $LGPC$ préserve les formules bégayement invariantes

Étant donné une formule φ de LTL bégayement invariante, on peut en déduire l'ensemble $AP_{Obs} \subseteq AP$ caractéristique de cette formule, on écrit alors $\varphi \in LTL(AP_{Obs})$. On peut donc en déduire l'ensemble $T_{Obs} \subseteq T$ des transitions qui peuvent modifier l'évaluation de la formule. Supposons que l'on construise alors un graphe (appelé $LGPC$) vérifiant les propriétés données dans la Définition 65, il nous faut alors montrer que la formule est préservée, i.e. que cette formule est satisfaite sur le graphe complet si et seulement si elle est satisfaite sur le graphe réduit $LGPC$.

Pour montrer qu'un $LGPC$ est bégayement équivalent au $\mathcal{STE}$ exhautif (par rapport à T_{Obs}), nous allons montrer qu'il préserve toutes les séquences observables. Une telle démonstration se décompose en deux étapes(voir définition 59) :

- D'une part il faut montrer que pour toute séquence d'exécution du $LGPC$ il existe une séquence dans le STE global qui ait la même séquence de transitions observables
- et d'autre part, que pour toute séquence du STE il existe une séquence d'exécution dans le $LGPC$ qui ait la même séquence de transitions observables

Chacune de ces parties est démontrée dans les sous-sections suivantes.

4.1 Toutes les séquences observables...

Considérons une séquence d'états $\sigma = s_0.s_1 \ldots$ de STE. Donc il existe $w = t_1.t_2 \ldots \in T^\omega$ tel que $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots$.
D'après (3) de la définition 65, il existe une séquence de pas $\Pi \in {T_R}^\omega$ telle que $f_{Obs}(w) = I\!F_{Obs}(\Pi)$ et $s \overset{\Pi_{\omega,\omega'}}{\rightarrow_R}$, donc il existe dans STE_R une séquence ayant même séquence observable.

4.2 ...et aucune séquence observable en plus.

Considérons une séquence d'états $\Gamma = e_0.e_1.e_2 \ldots$ dans le $LGPC$. Donc il existe une séquence de pas (finie ou infinie) $\Pi = \pi_1.\pi_2 \ldots$ telle que $e_0 \overset{\pi_1}{\rightarrow_R} e_1 \overset{\pi_2}{\rightarrow_R} \ldots$. D'après la construction des pas, toute séquence définie par $w = Seq(\pi_1).Seq(\pi_2).Seq(\pi_3) \ldots$ est une séquence de transitions du STE. Si $w = t_1.t_2.t_3 \ldots$ on en déduit $\sigma = s_0.s_1.s_3 \ldots$ tel que $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots$. On en déduit :

$$\begin{aligned} I\!F_{Obs}(\Pi) &= I\!F_{Obs}(\pi_1).I\!F_{Obs}(\pi_2).I\!F_{Obs}(\pi_3) \ldots \\ &= f_{Obs}(Seq(\pi_1)).f_{Obs}(Seq(\pi_2)).f_{Obs}(Seq(\pi_3)) \ldots \ . \\ &= f_{Obs}(w) \end{aligned}$$

Donc on obtient bien w une séquence de transtions du STE telle que $f_{Obs}(w) = I\!F_{Obs}(\Pi)$.

5 Algorithme de construction d'un $LGPC$

Un algorithme pour calculer un $LGPC$ est donné dans la table IV.1. L'algorithme est implicitement paramétré par une relation d'indépendance $\wr$ et un ensemble de transitions observables T_{Obs}. Les transitions sensibilisées sont divisées en deux sous-ensembles grâce aux fonctions T_U et T_M, avec les mêmes critères que ceux utilisés dans la construction des GPC. On supprime alors toutes les transitions observables qui pourraient être dans l'ensemble T_m et également toutes les transitions en conflit avec ces transitions observables ($[\#](T_{Obs}) \cap E$).

Finalement, T'_u contient les transitions à explorer d'une façon standard et T'_m contient celles dont l'exploration sera menée par des pas. Ces transitions seront nommées « fusionnables » dans la suite. $\Pi_{T'_M}$ est l'ensemble des pas de transitions, construit par la fonction Π.

Pour définir T_U et T_M on peut utiliser les définitions données pour les graphes de pas de conflits croisés.

develop_LGPC(E):
$T_u \leftarrow T_U(E, \iota)$
$T_m \leftarrow T_M(E, \iota)$
$T'_u \leftarrow T_u \cup (T_m \cap [\#](T_{Obs}) \cap E)$
$T'_m \leftarrow T_m \backslash [\#](T_{Obs})$
$develop\_exhaustive(T'_u)$
$\Pi_{T'_m} \leftarrow \Pi(T'_m, \iota)$
$develop\_by\_step(\Pi_{T'_m})$

Table IV.1: Algorithme de construction d'un $LGPC$

Dans l'algorithme précédent, les transitions en conflit avec une transition observable ont été exclues des pas et mises dans l'ensemble T_u. Nous expliquons ce choix sur l'exemple du réseau de Petri de la figure IV.6, dans lequel les transitions observables sont O_1 et O_2. Si on construit un Graphe de Pas couvrant en retirant uniquement les transitions observables de l'ensemble des transitions fusionnables, on obtient le graphe de la figure IV.6 à droite. La séquence $E_2.O_2.O_1$ réalisable dans le réseau de Petri n'est pas couverte par ce graphe de pas, par conséquent la propriété (3) de la Définition 65 n'est pas vérifiée par un tel graphe. Par exemple considérons une formule LTL qui exprime le fait que le marquage p_2, p_5 n'est pas accessible : $\varphi = []\neg(p_2 \wedge p_5)$. Donc $AP_{Obs} = \{p_2, p_5\}$ d'où $T_{Obs} = \{O_1, O_2\}$ ce sont effectivement les deux transitions qui peuvent changer les valuations de AP_{Obs}. La formule φ est satisfaite sur le graphe construit Figure IV.6, alors que le marquage p_2, p_5 est effectivement accessible et donc la formule fausse.

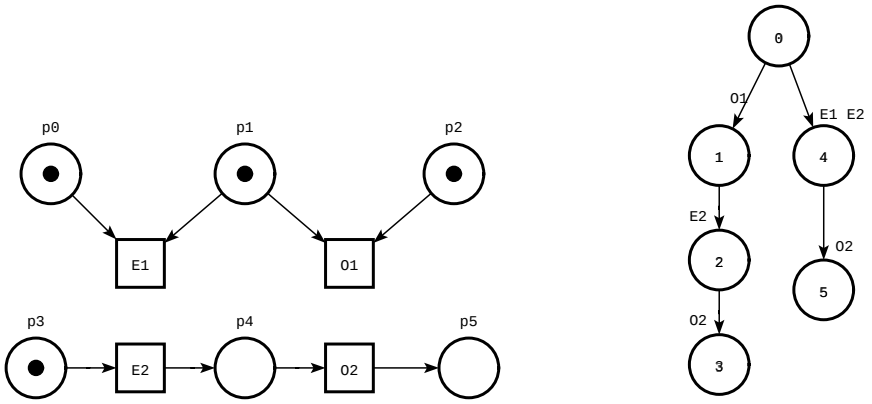


Figure IV.6: Exemple où on n'exclue pas $[\#](T_{Obs})$ mais uniquement T_{Obs}

Cet exemple permet d'expliquer nos choix, dans la section suivante nous démontrons que nos choix sont suffisants pour obtenir un $LGPC$.

6 Démonstration que l'algorithme construit un $LGPC$

Nous devons montrer que l'algorithme défini dans la section précédente permet bien de construire un $LGPC$. Par conséquent nous avons à vérifier que chaque condition de la définition 65 d'un $LGPC$ est respectée.

La condition (1) est prouvée dans [VAM96] puisque l'algorithme vérifie les propriétés définies

dans [VAM96].

La condition (2) est trivialement vérifiée car notre algorithme isole toutes les transitions observables pour les explorer de façon classique, donc les transitions observables se retrouvent dans des pas de taille 1 (une seule transition ; la transition observable).

La vérification de la condition (3) est plus technique. Il nous faut montrer qu'étant donnée une séquence w du $\mathcal{STE}$ il peut être trouvé une séquence w' et une séquence de pas Π du $\mathcal{STE}_R$ qui vérifie les propriétés : $[w.w'] = [\Pi]$ et $f_{Obs}(w) = I\!F_{Obs}(\Pi)$.

Pour cela nous allons introduire un opérateur de normalisation qui va nous permettre de construire le pas Π et w' de façon récursive.

6.1 Opérateur de normalisation

Pour construire cette suite de pas, nous allons utiliser une fonction de normalisation qui va extraire un pas, ou un préfixe d'un pas de la séquence w. La séquence Π_{n+1} sera définie en ajoutant à la séquence Π_n le pas donné par l'opérateur de normalisation N.

Cet opérateur extrait d'une séquence de transitions, le pas maximal ou le préfixe maximal d'un pas. L'opérateur N est défini en fonction de la relation de conflit choisie et de l'ensemble T_{Obs} des transitions observables.

Définition 66 *N est une fonction de $S \times Step(T) \times T^* \times T^\omega \mapsto Step(T) \times T^\omega$ définie par :*

$$N(s, E, w, \epsilon) = (E, w)$$
$$N(s, E, w_1, w_2) = (E, w_1.w_2) \quad si\ E \in \Pi_{T'_m}$$

$$N(s, E, w_1, t.w') = \begin{cases} N(s, E \cup \{t\}, w_1, w') & si\ t \in T'_m, \\ & t \wr w_1\ et\ E \cup \{t\} \in Step(T, \#^C, s)\ et\ t \notin E \\ (\{t\}, \mathcal{E}.w_1.w') & si\ t \in T'_u, \\ & t \wr w_1\ et\ \mathcal{E} \in Seq(E) \\ N(s, E, w_1.t, w') & sinon \end{cases}$$

Lemme 5 *Soit $s \in S$ et $w \in T^\omega$ tel que $s \xrightarrow{w}$, $N(s, \emptyset, \epsilon, w) = (E, w')$. Soit $F \in \Pi(T'_M)$. Alors :*

1. $s \overset{E}{\twoheadrightarrow} s_1 \xrightarrow{w'}$
2. $f_{Obs}(w) = I\!F_{Obs}(E).f_{Obs}(w')$
3. $E \subset F \Rightarrow (F \setminus E) \wr w'$ *et* $s \overset{F}{\twoheadrightarrow} s_1 \xrightarrow{w'}$
4. $E \subset F \Rightarrow f_{Obs}(w) = f_{Obs}(w')$.
5. $w = t.v \Rightarrow t \in E$

Démonstration

1. Découle directement de la construction de E par l'opérateur de normalisation, qui ne fait remonter que des transitions indépendantes de w_1 à chaque étape.
2. Si $E \cap T_{Obs} \neq \emptyset$ alors $E = \{t\}$ avec $t \in T_{Obs}$ sinon $I\!F_{Obs}(E) = \epsilon$ et donc $f_{Obs}(w) = I\!F_{Obs}(E).f_{Obs}(w')$.

3. Soit $s \in S, w, w' \in T^\omega$ tel que $s \xrightarrow{w}$, $N(s, \emptyset, \epsilon, w) = (E, w')$ et $F \in \Pi_{T'_M}$ tel que $E \subset F$. On prouve par l'absurde que $F \setminus E \wr w'$:
   Soit $R = F \setminus E$. Supposons qu'il existe $t \in ||w'||$[1] tel que $\exists t_r \in R$ avec $t \# t_r$, $w' = w_1.t.w_2$, $t_r \wr w_1$.
   Alors on a $t \wr w_1$ **et** $t \in T'_M$. $t \wr w_1$ car $t_r \wr w_1$, $t \# t_r$ et $\#$ est transitif. $t \in T'_M$
   Il y a deux raisons possibles au fait que la transition t n'ait pas été ajoutée à E par la fonction de normalisation N : soit $t \in E$ et de plus $t \in F$ alors $t_r \in F$, ce qui amène à la contradiction car $t_r \# t$. Ou $\exists\ t_{conflict} \in E$ tel que $t_{conflict} \# t$. Alors $t_r \# t_{conflict}$ et t_r et $t_{conflict}$ sont des transitions de F ce qui amène aussi à une contradiction.
4. car $I\!F_{Obs}(F) = I\!F_{Obs}(E) = \epsilon$ et (2).
5. Cette propriété vient de $N(s, \emptyset, \epsilon, t.w') = N(s, \{t\}, \epsilon, w')$. Donc si la première transition est dans T'_u alors $E = \{t\}$. Sinon E est un pas ou une partie d'un pas contenant la transition t.

6.2 Construction de la séquence de pas

L'objectif de cette section est de prouver que le graphe de pas construit vérifie bien la condition (3) de la définition 65 d'un $LGPC$.

Soit w un séquence du $\mathcal{STE}$. Il nous faut construire une séquence w' et une séquence de pas Π de $\mathcal{STE}_R$ qui vérifie les propriétés : $[w.w'] = [\Pi]$ et $f_{Obs}(w) = I\!F_{Obs}(\Pi)$.

On considère les suites suivantes :

- La suite de pas (π_i)
- La suite de séquences de pas (Π_i) telle que $\Pi_i = \pi_1 \dots \pi_i$.
- La suite de séquences de transitions (w_i) telle que $w_0 = w$.

Ces suites sont définies récursivement par : étant donné Π_n et w_n tels que $s_0 \xrightarrow{\Pi_n} s_n$, on définit w_{n+1} et Π_{n+1} (et donc π_{n+1}) à l'aide de l'opérateur de normalisation : $(E, w_{n+1}) = N(s_n, \emptyset, \epsilon, w_n)$.

- Si E est un pas ou $E = \{t\}$ avec $t \in T'_u$, alors $\pi_{n+1} = E$ **et donc** $\Pi_{n+1} = \Pi_n.E$
- Sinon, d'après le lemme 5 (3), il existe un pas F tel que $E \subseteq F$ et $(F \setminus E) \wr w_{n+1}$. On pose donc $\pi_{n+1} = F$ **et donc** $\Pi_{n+1} = \Pi_n.F$

Soit

$$\lim_{n \mapsto \infty} \Pi_n = \Pi$$

La difficulté réside alors dans le passage à la limite, où il faut montrer d'une part que Π appartient bien à $\mathcal{STE}_R$ et d'autre part $f_{Obs}(w) = I\!F_{Obs}(\Pi)$.

Le lemme 6 permet de montrer que Π appartient bien au $LGPC$. Le lemme 7 permet alors de montrer que la construction ne va pas laisser indéfiniment une transition observable dans w_i et donc finalement qu'à tout préfixe observable de w correspond un Π_i ayant la même séquence de transitions observables (lemme 8). On peut alors conclure dans le lemme 9 la propriété voulue.

Lemme 6 *Soit Π la séquence de pas construite comme la limite de la suite infinie Π_i, alors Π appartient au $LGPC$.*

Démonstration La démonstration est faite par induction sur la longueur du préfixe Π_i de Π. Le cas d'initialisation est que Π_0 est le mot vide, donc on reste dans l'état initial qui est un état initial dans le $LGPC$. Pour l'induction, on suppose que Π_i est dans le $LGPC$. Alors notons que Π_{i+1} est obtenu à partir de Π_i en ajoutant un pas ou une seule transition de l'ensemble T'_u.

[1] $||w||$ donne le support d'une séquence. Exemple : $||t_1.t_2.t_1|| = \{t_1; t_2\}$

Lemme 7 *Soit t_γ la première transition visible de w_i et $prefix_{t_\gamma}(w_i)$ la séquence maximale de w_i qui ne contient pas t_γ. Alors :*

- Si t_γ est la première transition de w_i alors $\pi_{i+1} = \{t_\gamma\}$,

- Si t_γ n'est pas la première transition de w_i, alors t_γ est la première transition visible de w_{i+1} et $\pi_{i+1} \in I\!F_{Obs}{}^{-1}(\epsilon)$ et $prefix_{t_\gamma}(w_{i+1}) \subset prefix_{t_\gamma}(w_i)$[2]

Démonstration Le premier cas du lemme est prouvé par la définition de l'opérateur de normalisation. Si la première transition est visible alors il retourne cette transition.

Le second cas se démontre d'une part par la propriété 5 du lemme de normalisation qui montre que la première transition de la séquence est forcément prise et d'autre part qu'aucune autre transition observable ne peut être prise avant t_γ puisque les transitions prises pour l'ensemble E doivent appartenir à T'_m.

Lemme 8 *Soit $v \in Prefix(f_{Obs}(w))$, alors il existe i tel que $v = I\!F_{Obs}(\Pi_i)$*

Démonstration La démonstration est faite par induction sur la longueur de v. Le cas d'initialisation $|v| = 0$ est trivialement vrai. Dans le cas inductif, on doit prouver que si $v.t$ est un préfixe de $f_{Obs}(w)$ et qu'il y a une séquence Π_i telle que $I\!F_{Obs}(\Pi_i) = v$, alors il existe une séquence Π_j avec $j > i$ telle que $I\!F_{Obs}(\Pi_j) = v.t$. On a besoin de montrer que t sera éventuellement ajouté à Π_j pour un $j > i$ et qu'aucune autre transition visible ne sera ajoutée à Π_k pour $i < k < j$. D'après la définition de l'opérateur de normalisation N, une transition visible est ajoutée à la fin de Π_k pour construire Π_{k+1} seulement si elle apparaît comme la première transition de w_k.

Le lemme 7 montre que la transition t reste la première transition visible dans les séquences successives w_k après w_i jusqu'à ce qu'elle soit mise dans un pas π_j. La séquence de transitions avant t se décale dans chaque w_k et la première transition de w_k est supprimée et mise dans le pas π_{k+1}. Ainsi, t est ajouté à partir d'une séquence Π_j.

Lemme 9

$$I\!F_{Obs}(\Pi) = f_{Obs}(w)$$

Démonstration D'après le lemme 8, tout préfixe observable de w est un préfixe observable de Π, comme par ailleurs d'après la définition de l'opérateur de normalisation il est évident que Π ne contient pas plus de transitions observables que w, alors $I\!F_{Obs}(\Pi) = f_{Obs}(w)$.

Ce dernier lemme nous permet de conclure que l'algorithme de la section 5 construit bien un $LGPC$.

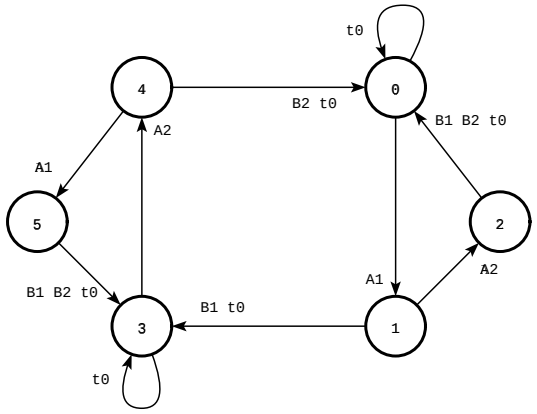
7 Exemples d'utilisation

La Figure IV.7 présente le $LGPC$ construit à partir du modèle du scheduler modifié pris en exemple dans la section 1. Cette fois-ci le graphe obtenu ne satisfait pas la formule $\varphi = [] <> M_1$. En effet la variable propositionnelle M_1 est vraie uniquement dans les états 0 et 2 donc une séquence qui reste indéfiniment dans les états 3, 4 ou 5 après un certain préfixe ne satisfait plus jamais M_1.

Nous avons comparé la taille du graphe construit par la technique développée ici à la taille du graphe exhaustif, mais également au graphe réduit obtenu par la méthode des ensembles amples puisqu'ils préservent également les propriétés de LTL. La Table IV.8 récapitule les résultats obtenus. Il est important de noter que la taille du graphe dépend fortement des transitions observables, les transitions observées dans ces exemples sont les suivantes :

Token Ring :

[2] Avec $u \subset v$ ssi u peut s'obtenir à partir de v en supprimant des transitions.



État	Marquage
0	$M_1\ s_{12}\ s_{22}\ P_0$
1	$M_2\ s_{11}\ s_{22}\ P_0$
2	$M_1\ s_{11}\ s_{21}\ P_0$
3	$M_2\ s_{12}\ s_{22}\ P_0$
4	$M_1\ s_{12}\ s_{21}\ P_0$
5	$M_2\ s_{11}\ s_{21}\ P_0$

Figure IV.7: $LGPC$ pour le scheduler de Milner modifié

– $T_{Obs}1$ correspond à une observation locale à chaque station :

$$T_{Obs}1 = \{1_{work}, 1_{rel}, 2_{work}, 2_{rel} \dots 10_{work}, 10_{rel}\}$$

– $T_{Obs}2$ correspond à une observation locale uniquement sur la première station

$$T_{Obs}2 = \{1_{work}, 1_{rel}\}$$

Scheduler : $T_{Obs} = \{A_1, A_2 \dots A_{10}\}$.
Base de données : Observation du service global

$$T_{Obs} = \{1\_send, 1\_ra, 2\_send, 2\_ra, \dots 100\_send, 100\_ra\}$$

Piscine : L'observation est $T_{Obs} = \{T_1, T_6\}$
Buffers asynchrones : Observation de l'envoi sur le premier buffer et observation de la réception sur le dernier. $T_{Obs} = \{p0\_to\_p1, p8\_to\_p9\}$. Cette observation permet de vérifier une propriété du genre $\varphi = [](send\_p0 \Rightarrow <> compute\_p9)$ (S'il y a un envoi du premier buffer alors il y a une réception par le dernier buffer).

Nom du modèle	Exhaustif	$LGPC$	Ample
Token Ring 10, $T_{Obs}1$	35 840	52	32 493
Token Ring 12, $T_{Obs}1$	172 032	62	156 927
Token Ring 10, $T_{Obs}2$	7 168	42	48
Scheduler 10 , T_{Obs}	10 240	110	38
Scheduler 100, T_{Obs}	Explosion	10 100	398
Base de Données 4, $T_{Obs}1$	109	13	29
Base de Données 8, $T_{Obs}1$	17 497	25	121
Base de Données 100, $T_{Obs}1$	Explosion	301	Explosion
Piscine 10, $T_{Obs}1$	7 006	4 135	6 847
Buffers asynchrones 9, T_{Obs}	8 748	688	1 745

Figure IV.8: Comparaison de $LGPC$ et des ensembles amples

Remarquons que pour les exemples du Token Ring avec $T_{Obs}1$ ou de la Base de données, les ensembles amples permettent de réduire la taille du graphe mais n'ont pas supprimé l'explosion

combinatoire tandis que la méthode des $LGPC$ permet d'obtenir un graphe de taille linéaire au nombre de stations.

Sur d'autres exemples l'inverse peut se produire ; par exemple dans le cas du scheduler de Milner passer de 10 à 100 sites multiplie la taille du graphe exhaustif par $1,24 \times 10^{28}$. La taille du graphe $LGPC$ est quand à elle multipliée par 100 : l'augmentation est donc plus faible mais plus élevée qu'avec la technique des ensembles amples qui est de 10 sur cet exemple.

Enfin notons que les réductions obtenues par les ensembles amples et la technique $LGPC$ sont faibles sur l'exemple de la piscine.

8 Expérimentation, comparaison des constructions

Dans la partie précédente les approches $LGPC$ et ensembles amples sont comparées sur la réduction obtenue sur le graphe de comportement. Comme nous l'avons vu au chapitre I, la vérification de formules LTL est constituées de différentes étapes. En particulier la construction d'un graphe synchronisé entre l'automate de Büchi de la formule et le graphe de comportement. Nous avons souhaité vérifier si la réduction obtenue sur le graphe de comportements permettait la construction d'un graphe synchronisé plus petit dans la pratique.

La table IV.2 permet de comparer les tailles des automates synchronisés construits d'une part dans le cas d'une synchronisation avec le graphe complet et d'autre part d'une synchronisation avec un graphe réduit. Il est intéressant d'observer expérimentalement que la réduction obtenue sur le graphe réduit du comportement du système est proportionnelle à la réduction obtenue sur le graphe synchronisé. Cette remarque permet donc de légitimer les comparaisons faites directement sur la taille des graphes réduits et non pas sur des graphes synchronisés. De plus cela permet d'avoir une comparaison qui ne dépend pas de la formule utilisée mais de l'ensemble des variables propositionnelles utilisées.

Modèle	STE Complet	STE Réduit	Büchi synchro. complet	Büchi synchro. réduit
Scheduler 10	10 240	15	20 186	26
BDD Jensen	1 459	19	1 459	19
Naimi Trehel	202 500	119 695	202 500	119 695
Piscine	38 759	842	38 760	843

Table IV.2: Comparaison de la taille des graphes construits

Pour vérifier si un marquage est accessible, le graphe synchronisé aura exactement le même nombre d'états que le graphe de comportements, puisque vérifier la formule consiste à chercher le marquage dans tous les états accessibles.

9 Conclusion

L'approche présentée dans ce chapitre permet la construction d'un graphe réduit préservant toutes les formules bégayement invariantes de $LTL(AP_{Obs})$. Cette technique, basée sur la technique des pas couvrants, permet la préservation de formules de la logique linéaire. Pour cela on déduit les transitions « importantes » par rapport aux variables propositionnelles utilisées dans la formule. Ces transitions « importantes » ne sont pas utilisées pour la construction de pas.

L'approche définie obtient de bons résultats en permettant de réduire des graphes que la méthode des ensembles amples ne permettait pas de réduire. La taille et la réduction du graphe construit dépendent du modèle mais également des variables propositionnelles utilisées dans la

formule. La complexité de la formule n'intervient pas : deux formules bégayement invariantes utilisant les mêmes variables propositionnelles nécessiteront le même graphe $LGPC$. Par contre la complexité de la formule LTL interviendra au moment du model checking de la formule sur le graphe réduit : la taille de l'automate de Büchi représentant la négation de la formule sera plus ou moins importante (et donc le graphe synchronisé aussi) en fonction de la complexité de la formule.

Remarquons enfin que l'approche $LGPC$ est compatible avec une vérification dite « au vol », consistant à vérifier la propriété au fur et à mesure de la construction du graphe et ainsi de n'explorer parfois qu'un sous-graphe pour trouver un contre-exemple à la propriété. Les tailles données dans les tableaux précédents n'utilisent pas la vérification « au vol » pour permettre la comparaison entre les différentes techniques.

Chapter V

Graphe de pas persistants

Comme nous l'avons vu dans l'état de l'art, deux directions sont classiquement abordées pour réduire l'explosion combinatoire : le choix d'un sous-ensemble de transitions à explorer ou la construction de pas pour regrouper le tir de transitions. Les résultats obtenus par les différentes méthodes de réduction montrent que la taille du graphe réduit obtenu dépend de la technique mais également du modèle. La table V.1 donne quelques exemples qui montrent que parfois le GPC est beaucoup plus performant (exemple du Scheduler) alors que sur d'autres exemples le graphe persistant parvient à construire un graphe plus petit (exemple des philosophes). L'idée du travail développé dans ce chapitre, est de faire collaborer deux techniques différentes pour en cumuler les bénéfices. Le cadre choisi a été la recherche d'états de blocages dans le système. Une seconde motivation est l'affaiblissement des graphes de pas couvrants : les GPC préservent les états de blocages mais préservent également les traces maximales. Par conséquent en affaiblissant la classe de propriétés préservées on espère construire des graphes plus petits.

L'objectif est de faire collaborer la technique des graphes de pas couvrants avec la technique des ensembles persistants. Le nouvel algorithme obtenu est appelé dans la suite « graphe de pas persistant » et noté GPP.

Modèle	Exhaustif	GP_{min}	GPC
Scheduler 300	$2^n * n = 6.10^{92}$	1 394	301
Philosophe 8	103 681	233	31 231

Table V.1: Évaluation des Graphes Persistants et Graphes de Pas Couvrants

1 Définition d'un graphe de pas persistant

1.1 Construction d'un graphe de pas persistant

Pour comprendre intuitivement à quoi correspond un graphe de pas persistant il faut reprendre l'idée intuitive de la réduction opérée par les méthodes la constituant. Le graphe de pas utilise l'indépendance des transitions pour ne faire qu'un seul pas « entre l'état de départ et l'état d'arrivée » permettant ainsi d'éviter tous les états intermédiaires. Les ensembles persistants

sont eux utilisés pour ne pas explorer toutes les transitions sensibilisées mais uniquement un sous-ensemble de celles-ci, permettant de limiter ainsi le nombre de successeurs visités.

L'approche GPP qui combine les deux, utilise les ensembles persistants pour ne pas envisager l'exploration de toutes les transitions et les pas pour grouper autant que possible les transitions que l'on a décidé d'explorer. L'algorithme, défini dans la table V.2, suit cette approche intuitive ; après avoir partitionné l'ensemble des transitions tirables dans les ensembles T_U et T_M en fonction de la relation de conflit, on choisit un ensemble persistant sur l'ensemble des transitions « fusionnables » T_M. Les pas sont alors construits par la fonction Π sur ce sous-ensemble de transitions.

```
develop_GPP(E):
        T_u ← T_U(E, ι)
        T_m ← T_M(E, ι)
        If T_m = ∅ Then
                develop_exhaustive(T_u)
        Else
                P ← E_P(T_m)      /* Choix d'un ensemble Persistant */
                Π_P ← Π(P, ι) /* Choix des Pas */
                develop_by_step(Π_P)
```

Table V.2: Algorithme Générique de Graphe de Pas Persistant

Cet algorithme est semblable à l'algorithme des graphes de pas couvrant (GPC) décrit dans la Table III.1. Les principales différences sont :

- l'algorithme GPP utilise une fonction de calcul d'ensembles persistants.
- Les transitions non fusionnables T_u ne sont pas toujours explorées. Elles ne sont effectivement tirées que lorsque $Enabled(s)$ n'admet pas d'ensemble persistant propre (i.e. $T_m = \emptyset$). Dans ce seul cas une exploration classique est effectuée. Intuitivement cela se justifie par le fait que si l'on trouve un ensemble persistant (et T_m est un ensemble persistant s'il est non vide) les transitions sensibilisées qui ne sont pas dans cet ensemble persistant n'ont pas besoin d'être explorées donc en particulier les transitions de T_u.
- Dans les GPC toutes les transitions de T_m sont tirées. Tandis que dans les GPP seul un sous-ensemble de T_m, défini par la fonction ensemble persistant $E_P()$, est considéré.

L'algorithme obtenu est générique dans le sens où deux fonctions peuvent être « instanciées » de façon libre : d'une part on peut choisir la fonction qui sélectionne un ensemble persistant parmi les transitions sensibilisées et d'autre part la fonction qui construit les pas sur un ensemble donné. Dans la suite nous allons utiliser ces paramètres de la méthode pour montrer de façon théorique les avantages de cette méthode. Plusieurs instances de cet algorithme sont testées sur des exemples concrets en essayant de montrer l'influence de ces paramètres.

1.2 Préservation des blocages

Cette méthode permet de construire un graphe réduit qui a les mêmes états de blocages que le graphe d'accessibilité exhaustif.

Proposition 2 *Les explorations GPP conservent les blocages.*

La démonstration de la proposition 2 est similaire à la démonstration faite pour les GPC dans la partie précédente : lemme de normalisation pour extraire des pas d'une séquence de transitions,

puis démonstration par récurrence sur la longueur de la séquence menant au blocage.

Sans donner les détails de la démonstration, une idée intuitive du déroulement de l'algorithme permet de « visualiser » la conservation des blocages. Un ensemble persistant est utilisé pour ne pas tirer toutes les transitions sensibilisées mais seulement un sous-ensemble de celles-ci, or on sait que choisir un sous-ensemble persistant ne fait pas perdre et ne rajoute pas d'états de blocage. Sur ce sous-ensemble on ne tire pas séquentiellement chaque transition mais au contraire des pas sont calculés enfin d'éviter tous les états intermédiaires, étant donné que les pas ne font qu'éviter des états intermédiaires qui conduisent au même état final, aucun blocage n'est ajouté ou supprimé là non plus. Par contre il faut être sûr qu'il n'existe pas une séquence différente qui puisse partir depuis un état intermédiaire et conduise à un état de blocage qui n'est pas pris en compte. Ce cas est typiquement le cas de confusion ; les transitions pour lesquelles cela peut arriver ont été mises dans l'ensemble T_u.

2 Comparaison des graphes de pas persistants

Nous allons faire une comparaison théorique entre d'une part la technique des graphes de pas persistants et d'autre part les ensembles persistants ou les graphes de pas couvrants. Cette comparaison permet de montrer que les graphes de pas persistants généralisent les ensembles persistants et améliorent les graphes de pas couvrants (pour la préservation des blocages). Nous donnons alors différentes instances possibles en fonction du choix des fonctions de sélection d'un ensemble persistant et de construction des pas. Enfin la dernière section présente des résultats expérimentaux permettant de comparer ces différentes techniques.

2.1 Étude des graphes de pas persistants

L'algorithme GPP est un algorithme général qui préserve les blocages et qui peut être instancié de différentes façons par la fonction utilisée pour le calcul de l'ensemble persistant[1] et pour le calcul des pas (Proposition 2). Dans la suite nous proposons différentes instances de l'algorithme et les comparons avec la technique des ensembles persistants et des pas couvrants.

2.1.1 Comparaison avec les ensembles persistants

Proposition 3 *GPP généralise GP*

Démonstration Pour démontrer que le GPP est une généralisation du GP, nous montrons que les graphes que nous pouvions obtenir avec la méthode des graphes persistants peuvent être obtenus avec la nouvelle méthode.

Considérons un graphe persistant G_1 construit par l'algorithme GP avec une fonction E_{P1}. Définissons une fonction « pas » qui ne construit que des pas réduits à une seule transition : $\Pi_1(E, \wr) = \{\{t\} \mid t \in E\}$. L'algorithme GPP défini avec les fonctions E_{P1} et Π_1 produit alors exactement le même graphe G_1. Par conséquent tout GP est un GPP.

Il est donc évident que si l'on choisit bien les fonctions $\Pi()$ et $E_P()$ le GPP peut produire des graphes plus petits que les graphes produits par GP. Cependant, l'algorithme GPP est plus général que l'algorithme GP : tout GPP n'est pas la contraction d'un GP.

Cette proposition montre que pour chaque algorithme de construction d'ensemble persistant, (i.e. pour chaque instance de l'algorithme GP) on peut définir une instance de GPP qui est au moins aussi bonne. Nous avons voulu savoir s'il était possible de trouver une instance de GPP

[1] Dans la mesure où elle calcule effectivement un ensemble persistant !

qui soit plus efficace que toutes les instances de GP. Malheureusement ce n'est pas le cas : compte tenu du non-déterminisme de l'algorithme GP, il est impossible d'avoir à coups sûr « autant de chance que lui ». Nous illustrons ce propos par le réseau de Petri de la Figure V.1. Ce réseau est constitué de composants non connectés (donc indépendantes) : $Loop$, $\mathcal{N}_0$, $\mathcal{N}_1$, $\ldots \mathcal{N}_n$.

La composante $Loop$ est simplement une boucle, tandis que la composante $\mathcal{N}_i$ est une chaîne de longueur k_i.

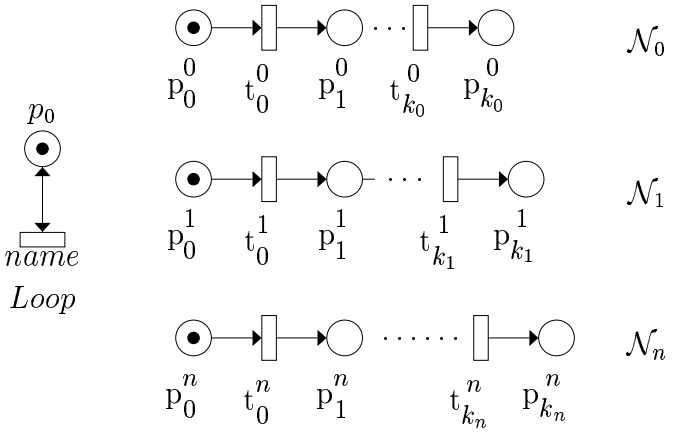


Figure V.1: Comparaison de GPP et GP_{min}

Différentes explorations par ensembles persistants minimaux GP_{min} peuvent être construites :
- L'exploration qui commence par la transition du composant $Loop$ comme ensemble persistant, est optimale avec seulement 1 état et 1 arc.
- L'exploration la plus catastrophique (exploration du composant $Loop$ en dernier) construit un graphe avec $\Sigma_{i=0}^{i=n} k_i + 1$ états.

Ces deux explorations sont du type GP_{min} mais conduisent à des résultats très différents à cause du non déterminisme de la construction. Avec l'approche GPP, soit on construit simplement un GP sans faire de pas et alors on aura les mêmes résultats que GP (i.e. « plus ou moins de chance »), soit on essaye de faire des pas et comme les chaînes sont indépendantes mais de longueurs différentes, le graphe construit a plus d'un état et donc n'est pas aussi bon que le « GP_{min} chanceux ».

2.1.2 Comparaison avec les graphes de pas couvrants

Proposition 4 *GPP améliore GPC (pour la préservation des blocages)*

Bien que GPP ne soit pas une généralisation de GPC (car un GPC n'est pas toujours un GPP), nous pouvons comparer les graphes obtenus par les deux types d'exploration. Dans l'exploration GPP, les transitions non fusionnables T_U ne sont pas toujours explorées. Cela a pour conséquence la perte de la propriété de couverture que nous avions pour les GPC mais permet de construire des graphes plus petits que ceux obtenus par l'exploration GPC ; en ce sens le GPP améliore les GPC pour la préservation des blocages.

2.2 Différentes instanciations

De la même façon que GP est un schéma général d'algorithme admettant différentes instanciations (GP_{min} est l'une d'entre elles), nous allons maintenant envisager différentes instanciations

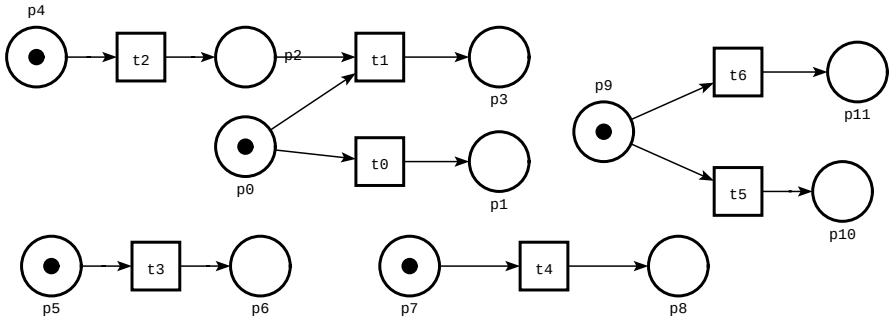


Figure V.2: Exemple de réseau de Petri

de GPP en choisissant des fonctions $E_P()$ et $\Pi()$ spécifiques. La première instance vise à minimiser le branchement et s'apparente à GP_{min}, la seconde favorisant les pas se rapproche de GPC et la troisième est de nature plus hybride.

Pour instancier cet algorithme, on peut s'intéresser plus particulièrement aux transitions libres de tout conflit[2]. En effet on sait que tout ensemble réduit à une transition libre de tout conflit est un ensemble persistant. Donc un ensemble composé de transitions libres de tout conflit est aussi un ensemble persistant. Par ailleurs ces transitions peuvent être fusionnées avec n'importe quelles autres transitions, en particulier toutes les transitions libres peuvent être mises dans un pas unique. Aussi dans la suite nous allons nous intéresser à l'ensemble F_E de toutes les transitions sensibilisées dans l'état s qui sont libres de tout conflit.

$$F_E(s) = \{t \in Enabled(s) \mid \#(t) = \{t\}\ \}$$

Pour illustrer les différentes constructions proposées, nous utilisons le réseau de Petri de la figure V.2. Le graphe de comportement exhaustif de ce réseau de Petri comporte 60 marquages et 160 transitions avec 4 états de blocage. L'algorithme des pas couvrants construit un graphe de 8 états présenté dans la figure V.3 qui met en évidence les 4 états de blocage (états 6,5,29 et 30). La numérotation des états dans ce graphe est faite par rapport aux numéros des états dans le graphe exhaustif. Le dessin du graphe exhaustif est donné en arrière plan de la Figure II.12. Le graphe exhaustif étant de taille importante pour être dessiné, nous proposons de représenter les graphes obtenus par les différentes instances en ne laissant en-arrière plan que les états présents dans le GPC.

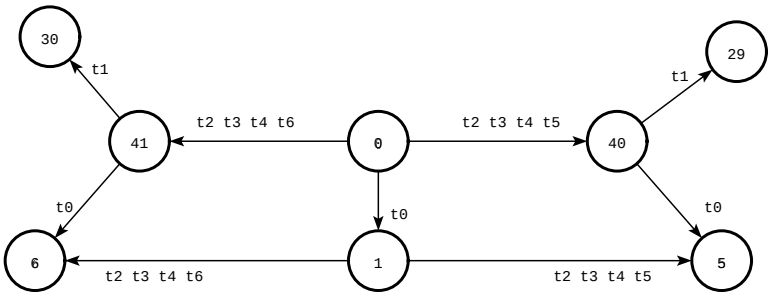


Figure V.3: Exploration GPC

[2] transitions pour lesquelles $\#(t) = \{t\}$

2.2.1 Instance GPP_{min}

Nous considérons un algorithme GPP particulier qui minimise le branchement. Cet algorithme sera noté GPP_{min} dans la suite. Les pas sont construits en utilisant l'orthoproduit π_C pour la fonction $\Pi()$.

La fonction $E_P()$ est définie par :

$$E_P(s) = \begin{cases} F_E(s) & \text{si } F_E(s) \neq \emptyset \\ \#(t_{min}) & \text{sinon} \end{cases}$$

avec $t_{min} \in Enabled(s)$ telle que $Card(\#(t))$ est minimal pour $t = t_{min}$.

Cette fonction $E_P()$ choisit un ensemble persistant minimal (donc minimise le branchement local) en sélectionnant éventuellement toutes les transitions libres de tout conflit lorsqu'il y en a.

Remarquons que l'heuristique utilisée par le GPP_{min} est la même que celle utilisée pour le GP_{min}. Notons également que (comme le GP_{min}) le GPP_{min} n'est pas déterministe (le choix entre différents ensembles persistants minimaux est arbitraire).

L'utilisation de GPP_{min} sur l'exemple de la Figure V.2 produit le graphe de la Figure V.4. Dans l'état s_0, les transitions t_2, t_3 et t_4 sont libres de tout conflit (donc $E_P(s_0) = F_E(s_0) = \{t_2, t_3, t_4\}$) et un seul pas contenant ces 3 transitions est construit. Dans les états s_{40} ou s_{41} on obtient $F_E = \emptyset$ et on choisit donc $E_P(s_{40}) = E_P(s_{41}) = [\#](t_0) = [\#](t_1) = \{t_0, t_1\}$.

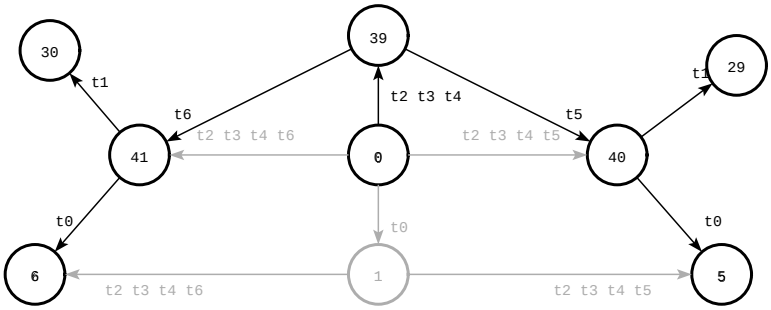


Figure V.4: Exploration GPP_{min}

Une exploration par GPP_{min} de l'exemple de la Figure V.1 est déterministe et conduit à un graphe ayant $Max\{k_i + 1 : i \in [1, n]\}$ états, ce qui représente la longueur de la chaîne la plus longue + 1. Cette valeur est très inférieure au pire des cas obtenu avec GP_{min}.

2.2.2 Instance $GP_{max}P$

Dans cette instance de GPP la priorité est donnée aux pas : on souhaite construire des pas les plus grand possibles afin d'éviter au maximum les états intermédiaires. Cet algorithme est noté $GP_{max}P$ dans la suite.

On définit la fonction persistant par $E_P(E) = E$ pour avoir E_P (l'ensemble sur lequel sont fait les pas) aussi grand que possible. Les pas sont calculés en utilisant l'orthoproduit π_C.

L'application de $GP_{max}P$ sur l'exemple de la Figure V.2 construit le graphe de la Figure V.5. Dans l'état initial, la transition t_0 est en conflit avec une transition non sensibilisée t_1, par conséquent elle est mise dans l'ensemble T_u qui n'est pas exploré puisque l'ensemble T_m n'est pas vide. Les transitions t_5 et t_6 sont en conflit et produisent donc deux pas différents. Dans les

états s_{40} et s_{41}, les transitions sensibilisées sont t_0 et t_1 et elles sont en conflit, donc plus aucun « pas » n'est possible.

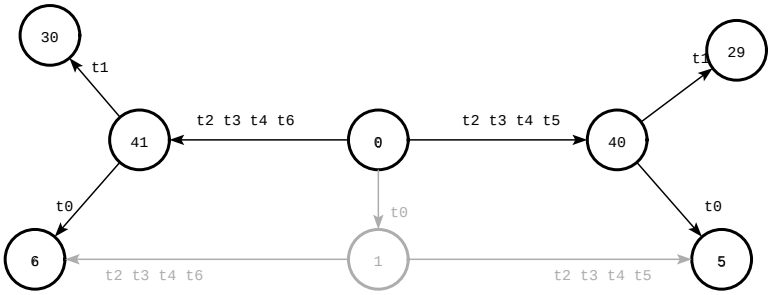


Figure V.5: Exploration $GP_{max}P$

Proposition 5 Le $GP_{max}P$ est toujours plus petit que le GPC

Démonstration Les algorithmes GPP exploitent le fait que l'ensemble T_M est un ensemble persistant lorsqu'il n'est pas vide et évitent ainsi l'exploration des transitions de T_U. Dans le cas particulier où la fonction persistant est la fonction identité (car T_m est un ensemble persistant) les pas sont construits sur l'ensemble T_m entier exactement comme dans l'algorithme GPC. Par conséquent le graphe obtenu est donc trivialement un sous graphe du GPC.

2.2.3 Instance $GPPH$

Les algorithmes GPP_{min} et $GP_{max}P$ diffèrent seulement par leur fonction de choix de l'ensemble persistant ; qui retourne l'ensemble minimal dans un cas et l'ensemble total des transitions sensibilisées dans le second cas. En ce sens, les deux instances ci-dessus sont les deux « extrêmes » ; l'une utilisant les ensembles persistants, l'autre utilisant les pas. Entre ces deux extrêmes nous pouvons trouver toutes les combinaisons possibles. La proposition 2 assure que toutes les solutions intermédiaires possibles préservent les blocages.

Le fait de choisir dans la première étape, systématiquement l'ensemble persistant le plus petit possible empêche de faire véritablement des pas dans la seconde étape de l'algorithme. Il faut donc trouver un compromis, ou plus exactement choisir plus « intelligemment » un ensemble persistant qui permette la construction de pas conséquents.

Nous proposons une instance « Hybride », noté $GPPH$ dans la suite, permettant de concilier les ensembles persistants et les pas. Cette instance $GPPH$ présente en outre l'avantage d'être déterministe dans le sens où le graphe ne dépend plus d'un choix arbitraire d'ensemble persistant.Dans cette instance $GPPH$, les pas sont construits en utilisant l'orthoproduit π_C pour la fonction $\Pi()$. La fonction permettant de calculer un ensemble persistant est définie par :

$$E_P(s) = \begin{cases} F_E(s) & \text{si } F_E(s) \neq \emptyset \\ \text{Enabled(s) - } F_E(s) & \text{sinon} \end{cases}$$

Le graphe construit pour l'exemple de la Figure V.2 est représenté Figure V.6. Comme pour GPP_{min} de la Figure V.4, toutes les transitions libres de tout conflit sont tirées dans l'état initial s_0 pour atteindre l'état s_{39} dans lequel $E_P(s_{39}) = Enabled(s_{39}) = \{t_0, t_1, t_5, t_6\}$ et $\Pi_C(\{\{t_0, t_1\}, \{t_5, t_6\}\}) = \{\{t_0, t_5\}, \{t_0, t_6\}, \{t_1, t_5\}, \{t_1, t_6\}\}$.

Les différentes instances possibles peuvent être représentées sur un plan à deux axes, selon si on privilégie les ensembles persistants ou les pas. La figure V.7 propose une telle représentation.

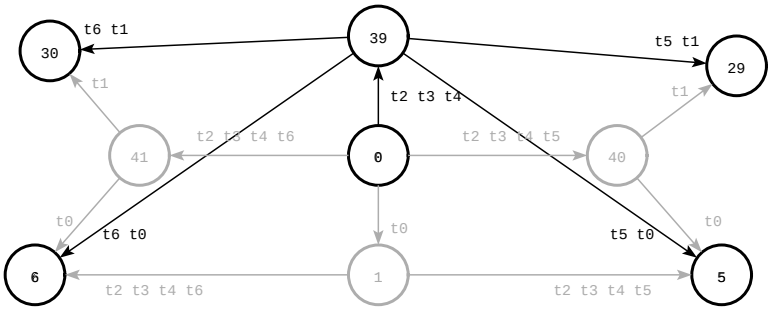


Figure V.6: Exploration $GPPH$

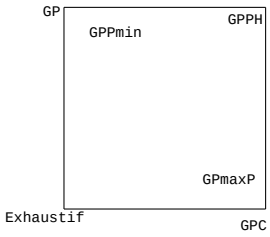


Figure V.7: Classement des différentes méthodes

2.3 Résultats expérimentaux

Les résultats comparatifs pour l'exemple de la Figure V.2 dont les différents graphes ont été donnés dans la section précédente sont résumés dans la Table V.3 en terme de nombre d'états et de transitions. Observons que sur cet exemple, toutes les stratégies GPP améliorent ou égalent les méthodes GP_{min} et GPC.

	$Exhaustive$	GP_{min}	GPC	GPP_{min}	$GP_{max}P$	$GPPH$
$States$	60	12	8	8	7	6
$Transitions$	160	11	9	7	6	5

Table V.3: Résumé des résultats

Ces algorithmes ont été appliqués à des modèles de taille significative et d'un intérêt pratique. Le premier modèle est une variation du scheduler de Milner à 300 sites [Mil89], le second est le modèle du dîner de philosophes avec 8 philosophes. Le troisième est le système de base de données présenté dans [Jen86] et utilisé comme exemple dans [Val89]. Le quatrième exemple est un modèle de token ring avec 10 stations [Cor96]. Le cinquième exemple modélise l'algorithme distribué d'exclusion mutuelle de Naimi-Trehel avec 5 sites [NT87]. Le sixième modèle est défini dans [ZDD93, CX97] et représente un système industriel. Le septième exemple, décrit des buffers asynchrones[Pel93]. Les différents résultats sont donnés dans la Table V.4. Les trois instances de GPP (GPP_{min}, $GP_{max}P$ et $GPPH$) donnent les mêmes résultats sur ces exemples[3] et sont notés

[3]Notre exemple d'étude montre que ce n'est pas toujours le cas

par $GPPH$. GP_{min} produit un graphe plus petit que GPC pour les exemples des Philosophes, Naimi-Trehel et le système industriel, tandis que le GPC est meilleur pour les autres exemples (scheduler de Milner, base de données, token ring et buffers asynchrones). Pour tous ces exemples, la technique $GPPH$ construit un graphe plus petit que les graphes GP_{min} et GPC. Remarquons que dans le cas de Naimi-Trehel on obtient une réduction (division par 4 ou 5), mais l'explosion combinatoire a toujours lieu que ce soit avec la technique des ensembles persistants, des pas, ou des pas persistants.

	Model	Exhaustive	GP_{min}	GPC	$GPPH$
1	Scheduler 300	$2^n * n \approx 6.10^{92}$	1 394	301	301
2	Philosophes 8	103 681	233	31 231	227
3	Base de données 10	196 831	191	31	31
4	Token Ring 10	35 840	99	52	51
5	Naimi-Trehel 5	202 500	40 006	52 681	40 001
6	Manufacturing system	2 034	455	979	360
7	Buffers asynchrones 7	972	37	12	12

Table V.4: Comparaison des algorithmes

Nous nous intéressons à un dernier exemple dit de la « piscine », présenté dans [BF99], dont le modèle est donné dans la Figure V.8 : La transition $T1$ représente l'arrivée d'un client qui prend une cabine libre (place $x6$) et se déshabille en place $x1$. Puis il prend un panier (transition $T2$). Les paniers libres sont comptés dans la place $x7$. Après avoir mis ses affaires dans le panier, le client libère la cabine (un jeton est remis en place $x6$) et va dans le bassin. À la sortie du bassin, transition $T4$ le client reprend une cabine libre. Après s'être changé il libère le panier, qui revient dans le stock des paniers libres (place $x7$). Puis il quitte la piscine et libère la cabine.

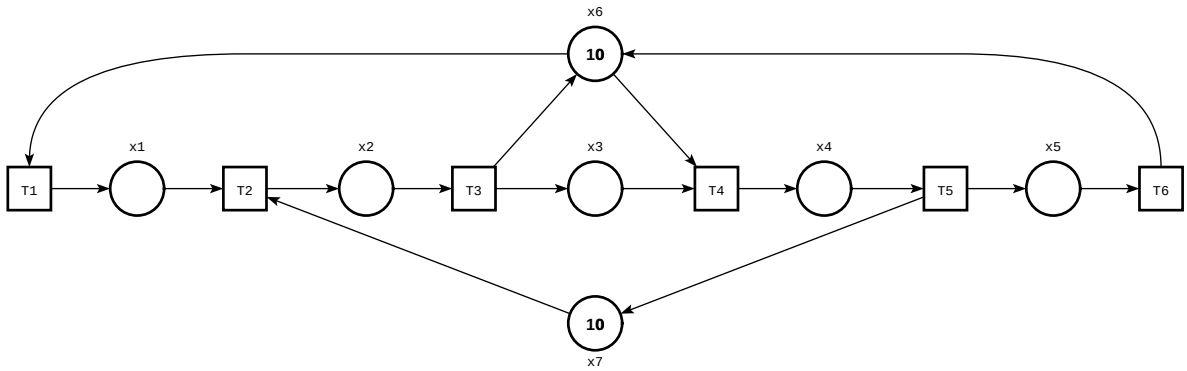


Figure V.8: La piscine

Les vérifications de ce modèle montrent que quelle que soit la valeur k (nombre de cabines et de paniers), un interblocage est toujours possible : une arrivée massive de clients occupe toutes les cabines, mais ces clients seront en attente d'un panier. Si plus aucun panier n'est libre, il faut attendre que certains d'entre eux se libèrent, mais dans le cas où tous les clients présents sont en train de se baigner, ils ne peuvent pas quitter la piscine car aucune cabine n'est disponible.

La Table V.5 donne la taille du graphe construit et le temps de calcul nécessaire à sa construction par les différentes techniques. Ces tests ont été réalisés sur une machine Sun Ultra 5, avec

2 Giga-octets de mémoire. Sur cet exemple, l'algorithme $GPPH$ améliore réellement à la fois le GP_{min} et le GPC. La construction du graphe exhaustif échoue pour $K = 50$ tandis que les explorations partielles y arrivent. GP_{min} échoue à partir de $K = 240$ tandis que le GPC échoue à partir de $K = 600$. La taille du graphe construit par l'algorithme $GPPH$ semble linéaire par rapport à K et permet la construction du graphe pour $K = 200\ 000$.

K	Exhaustif		GP_{min}		GPC		$GPPH$	
10	7 006	0:00:01	857	0:00:00	367	0:00:00	87	0:00:00
235	X	–	4 602 707	9:47:00	219 742	0:01:32	2 112	0:00:01
240	X	–	X	–	229 217	0:01:42	2 157	0:00:01
500	X	–	X	–	997 517	0:31:08	4 497	0:00:02
600	X	–	X	–	X	–	5 397	0:00:02
15 000	X	–	X	–	X	–	134 997	0:00:16
150 000	X	–	X	–	X	–	1 349 997	0:13:30
200 000	X	–	X	–	X	–	1 799 997	0:24:11

Table V.5: Evaluation sur l'exemple de la piscine. Légende : nombre d'états h :min :sec

3 Conclusion

Ce chapitre a étudié la coopération de deux techniques de réduction par « ordre partiel » : les explorations partielles basées sur les ensembles « persistants » et les « graphes de pas couvrants ». Un algorithme général d'exploration partielle mêlant ces deux approches et en cumulant les bénéfices est proposé : le GPP. Différentes instanciations de cet algorithme ont été comparées à l'utilisation séparée des ensembles persistants ou des pas.

Cette technique est une généralisation de ces méthodes. Les pas persistants peuvent « simuler » et améliorer les ensembles persistants et les pas couvrants. Bien que nous ne puissions pas exhiber de fonction persistante $E_P()$ générant « à tous les coups » un GPP meilleur que les ensembles persistants (car les ensembles persistants ne sont pas déterministes et le bénéfice de l'« ignoring problem » peut être perdu avec les pas), l'évaluation de ces constructions sur des exemples significatifs montre que ce but est souvent atteint dans la pratique avec l'instance $GPPH$.

Enfin, notons que (de la même manière que pour les deux techniques la combinant) la vérification au vol peut être utilisée conjointement à cette technique, en effet dès qu'un état de blocage est trouvé, on est sûr que cet état de blocage appartient bien au graphe exhaustif.

Le travail présenté dans ce chapitre a été intégré au logiciel *Tina* et publié dans [RVB02].

Chapter VI

Vérification de systèmes temporels

Différentes techniques ont été mises au point pour permettre la vérification de systèmes temporisés en construisant un graphe de comportement abstrait pour représenter de façon finie l'espace d'états (généralement infini) du système. Cependant la vérification de systèmes temporisés souffre d'une explosion du nombre d'états liée aux contraintes temporelles. Pour rendre ces techniques utilisables sur des systèmes de taille « importante » il est indispensable d'avoir des techniques de réduction de la taille du graphe construit. [Pag97] propose l'extension de techniques ordres-partiels (ensembles persistants et ensembles dormants) définies dans le cadre atemporel pour les utiliser avec des systèmes temporisés. Son travail montre que l'ajout de contraintes temporelles augmente les dépendances entres les transitions, ce qui limite les réductions possibles. [YS96] propose une technique de vérification de formules bégayement invariantes sur des systèmes temporels en utilisant les ordres partiels.

L'approche proposée ici est basée sur l'utilisation d'une sur-approximation du comportement qui préserve les propriétés de la logique LTL. La sur-approximation permet en général une procédure de décision semi-effective : lorsque la propriété est établie sur le graphe réduit (abstrait), on peut en déduire sa satisfaction sur le système, mais dans la négative on ne peut en général statuer. Lorsque la sur-approximation ne permet pas de conclure, nous utilisons les résultats partiels trouvés lors de cette première analyse pour « aider » la vérification de la propriété sur le comportement réel. Intuitivement cette seconde phase est une exploration temporelle « guidée » par les résultats de l'analyse de la sur-approximation. Cette seconde phase permet d'obtenir une procédure de décision effective.

Dans la première section, nous présentons une approche pour obtenir une sur-approximation du comportement et nous montrons comment cette approche peut être utilisée pour obtenir une technique semi-effective pour l'étude des états de blocage, ou pour la vérification d'une propriété LTL. Nous définissons alors une technique de décision effective en deux phases, par un guidage de l'exploration classique. Les limites et les extensions de cette approches sont alors présentées.

1 Analyse du comportement sur une sur-approximation

1.1 Construction d'une sur-approximation des séquences exécutables

Nous proposons d'obtenir une sur-approximation du comportement du système en relâchant certaines contraintes temporelles du réseau temporel qui le décrit. Pour cela on définit une nouvelle fonction intervalle statique I'_s telle que $\forall t, \quad I_s(t) \subseteq I'_s(t)$. On obtient donc $STE(R, I_s)$ le graphe d'accessibilité du réseau de Petri et $STE(R, I'_s)$ le graphe d'accessibilité de la sur-approximation. Dans la suite, nous montrons que les états de blocages peuvent être détectés sur

cette sur-approximation $STE(R, I'_s)$: si la sur-approximation ne contient pas d'état de blocage alors on peut en déduire que le système initial n'en a pas non plus.

Relâcher des contraintes temporelles conduit à des séquences de franchissement éventuellement impossibles dans le réseau initial : on obtient donc potentiellement un ensemble de marquages atemporels plus grand.

Propriété 18 *Inclusion des séquences*

Soit R un réseau de Petri, I_s et I'_s deux fonctions intervalles statiques telles que $\forall t, I_s(t) \subseteq I'_s(t)$. Soit $\longrightarrow$ la relation de tir dans le réseau de Petri (R, I_s) et $\longrightarrow'$ la relation de tir dans le réseau de Petri (R, I'_s).

Soit $\sigma = x_1.x_2 \ldots$ une séquence avec $\forall i, x_i \in T$ ou $x_i \in \mathbb{R}^{*+}$.

$$(m_0, I_s) \stackrel{\sigma}{\longrightarrow} (m, I) \Rightarrow (m_0, I'_s) \stackrel{\sigma}{\longrightarrow}' (m, I')$$

Avec $\forall t, I(t) \subseteq I'(t)$.

Démonstration

Montrons par récurrence sur la longueur de la séquence σ.

Trivial pour $|\sigma| = 0$.

Supposons que la propriété est vraie pour toute séquence de longueur inférieure ou égale à n.

Soit une séquence $\sigma = x_1.x_2 \ldots x_n.x_{n+1}$ de longueur $n+1$, avec $(m_0, I_s) \stackrel{\sigma}{\longrightarrow} (m_{n+1}, I_{n+1})$ dans $STE(R, I_s)$. Donc en particulier la séquence $\sigma_n = x_1.x_2 \ldots x_n$ préfixe de la séquence σ est tirable : $(m_0, I_s) \stackrel{\sigma_n}{\longrightarrow} (m_n, I_n)$ Donc d'après l'hypothèse de récurrence $(m_0, I'_s) \stackrel{\sigma_n}{\longrightarrow}' (m'_n, I'_n)$, avec $m'_n = m_n$ et $\forall t \in T, I_n(t) \subseteq I'_n(t)$. Deux cas se produisent :

- $x_{n+1} \in T$ (Transition discrète) : $(m_n, I_n) \stackrel{(x_{n+1})}{\longrightarrow} (m_{n+1}, I_{n+1})$ signifie que $Min(I_n(x_{n+1})) = 0$ d'où $Min(I'_n(x_{n+1})) = 0$ donc $(m_n, I'_n) \stackrel{(x_{n+1})}{\longrightarrow}' (m_{n+1}, I'_{n+1})$ et pour toute transition t nouvellement sensibilisées, on a $I_{n+1}(t) = I_s(t)$ et $I'_{n+1}(t) = I'_s(t)$ donc $I_{n+1}(t) \subseteq I'_{n+1}(t)$. Pour toute transition t qui reste sensibilisée $I_n(t) = I_{n+1}(t)$ et $I'_n(t) = I'_{n+1}(t)$, or $I_n(t) \subseteq I'_n(t)$ donc $I_{n+1}(t) \subseteq I'_{n+1}(t)$.
- $x_{n+1} \in \mathbb{R}^{*+}$ (Transition temps) : $(m_n, I_n) \stackrel{(x_{n+1})}{\longrightarrow} (m_{n+1}, I_{n+1})$ signifie que pour toute transition t sensibilisée par m_n on a $x_{n+1} \leq Max(I_n(t))$, donc $x_{n+1} \leq Max(I'_n(t))$ et $(m_n, I'_n) \stackrel{(x_{n+1})}{\longrightarrow}' (m_{n+1}, I'_{n+1})$. De plus le tir de la transition temps x_{n+1} « décale en coupant à zéro » les intervalles donc on a toujours $\forall t, I_{n+1}(t) \subseteq I'_{n+1}(t)$.

Le terme de sur-approximation du comportement est donc justifié par cette notion de sur-ensemble des séquences. Notons que la sur-approximation peut permettre d'atteindre un état $s = (m, I)$ dont le marquage m n'était même pas accessible dans le réseau de Petri initial.

Ce résultat peut sembler paradoxal puisque d'une part nous cherchons à réduire la taille du graphe construit et d'autre part nous construisons une sur-approximation contenant plus d'états temporels que le système initial. Cependant, comme nous le verrons par la suite, une abstraction est nécessaire pour représenter le comportement puisque le nombre d'états temporels accessibles est généralement infini. Nous parions donc sur le fait que le système moins contraint temporellement aura moins de comportements particuliers et permettra ainsi la construction d'une abstraction plus petite que l'abstraction construite pour le système initial.

1.2 Détection des blocages par une sur-approximation

L'inclusion de l'ensemble des états accessibles ne permet pas à elle seule de garantir la conservation des états de blocage. Il nous faut encore montrer qu'un état de blocage du graphe de comportement est un état de blocage dans la sur-approximation. Pour cela nous allons montrer que relâcher des contraintes temporelles ne supprime pas d'état de blocage. Avec le formalisme des réseaux de Petri temporels, la propriété de blocage ne dépend que du marquage (atemporel), d'où la propriété suivante :

Propriété 19 *Absence de blocage temporel dans un réseau de Petri temporel*
Soit (R, I_s) un réseau de Petri temporel. Soit R le réseau de Petri sous-jacent. Un état (m, H) du réseau de Petri temporel est un état de blocage ssi (m) est un état de blocage dans le réseau de Petri R.

Cette propriété découle directement de la sémantique d'un réseau de Petri temporel. Elle nous permet d'assurer que toute **séquence maximale du comportement sera toujours une séquence maximale dans la sur-approximation.**

Cette propriété est caractéristique des systèmes de transitions qui ont les intervalles de temps uniquement sur les transitions et des horloges locales relatives à l'instant de sensibilisation de la transition. Les systèmes de transitions temporisés définis dans [HMP91] ont la même propriété. Notons que cette propriété n'est pas vérifiée par exemple avec le formalisme des automates temporisés.

Considérons l'automate de la figure VI.1. L'horloge x est initialement nulle, la transition t_1 n'est tirable qu'après avoir laissé écouler 5 unités de temps, on se retrouve alors dans un état depuis lequel une seule transition est potentiellement tirable lorsque $x < 2$, ce qui ne sera jamais le cas, par conséquent le système est bloqué dans cet état. Si on construit le comportement du système sans tenir compte des contraintes temporelles on obtient le graphe à droite de la figure VI.1 : Il n'y a pas d'état de blocage dans la sur-approximation atemporelle, le blocage résulte donc des contraintes temporelles.

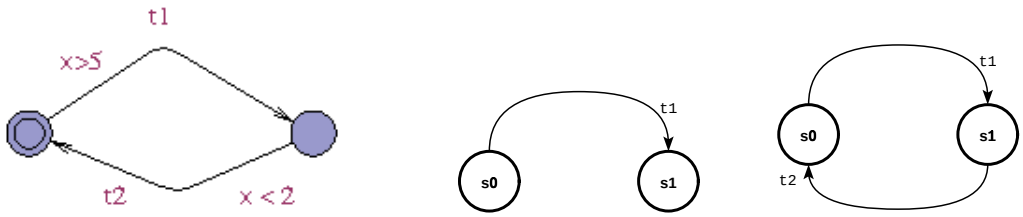


Figure VI.1: Automate temporisé - Comportement temporel - Surapproximation atemporelle du comportement

Notre objectif est d'exploiter cette propriété des réseaux de Petri temporels pour améliorer les performances de la vérification. Muni de la sur-approximation et de la propriété d'absence de blocage temporel, tout état de blocage se retrouve dans le comportement sur-approximé également comme un état de blocage. Donc si la sur-approximation ne comporte aucun état de blocage alors on peut conclure directement que le comportement du système ne comporte pas d'état de blocage.

1.3 Utilisation pratique de la sur-approximation

L'approche précédente a été définie sur l'ensemble des états temporels accessibles, cependant cet ensemble est généralement infini. Dans la pratique il est donc nécessaire de considérer une abstraction de ces états. Nous allons utiliser l'abstraction du graphe de classes qui permet de préserver les propriétés de LTL dont la construction est définie dans [BD91]. Cette abstraction peut donc être utilisée aussi bien sur le système initial que pour la sur-approximation.

À partir d'un réseau de Petri temporel de nombreuses sur-approximations peuvent être considérées en relâchant plus ou moins de contraintes. Si on considère une transition particulière avec un intervalle $|x;y|$[1], on peut remplacer cet intervalle pour tout intervalle $|x';y'| \subseteq |x;y|$. On peut relâcher par exemple la contrainte minimale qui empêche la transition d'être tirée tout de suite, on considérant l'intervalle $[0;y|$, il faudra alors veiller à ce que le réseau de Petri ne devienne pas non borné. On peut également relâcher la contrainte maximale, pour permettre de franchir la transition beaucoup plus tard, ou même jamais en considérant l'intervalle $|x;\infty[$. Cette sur-approximation risque de créer des états de divergence temporelle et/ou peuvent également rendre le réseau de Petri non borné si le fait de ne pas tirer la transition permet à des jetons de s'accumuler. On peut également considérer la sur-approximation maximale, qui relâche complètement les deux contraintes minimale et maximale en considérant l'intervalle $[0;\infty[$, ce qui correspond à considérer le comportement atemporel du système, à la nuance près qu'il n'y a pas de notion de divergence temporelle pour un comportement atemporel.

Les sur-approximations précédentes peuvent être appliquées brutalement sur toutes les transitions du système, ce qui permet d'obtenir des sur-approximations générales obtenues de façon automatique. Cependant une bonne connaissance du système modélisé est parfois nécessaire pour définir une sur-approximation plus fine en combinant les différentes possibilités sur différentes transitions, pour obtenir une sur-approximation efficace qui soit bornée et n'introduit pas de divergence qui empêcherait de prouver la propriété voulue.

Une sur-approximation rajoute des comportements et des états temporels. Par conséquent une sur-approximation est efficace uniquement si elle ajoute des comportements qui permettent à l'abstraction du graphe de classes de regrouper le maximum d'états temporels dans la même classe.

Vouloir absolument relâcher le maximum de contraintes par rapport à la propriété peut augmenter les comportements possibles et les marquages accessibles et donc faire augmenter la taille du graphe de classes, l'extrême étant caractérisé par un réseau non borné et donc un graphe de classes infini.

1.4 Exemple de sur-approximation atemporelle

Nous utilisons l'exemple des trains de Genrich[Gen87] représenté figure VI.2 : des trains circulent sur une boucle. La boucle est découpée en différents segments numérotés de 1 à n. Pour éviter les collisions, un train se trouvant sur le fragment numéro i ne peut aller sur le fragment $i+1$, uniquement si aucun train ne se trouve sur les segments $i+1$ et $i+2$, autrement dit un train « bloque » le segment qu'il occupe et le segment qui le précède. On propose le réseau de Petri de la figure VI.3 : le train présent dans la section 1 est signalé par un jeton dans la place $T1\_V1$ et occupe les segments $V1$ et $V7$ par conséquent les places correspondantes n'ont pas de jeton. Pour qu'il avance d'un segment par la transition $T1\_V1toV2$ il faut que la voie $V2$ soit disponible. Il libère alors la voie 7. Nous avons ajouté des contraintes temporelles pour signifier que le parcours d'une voie nécessite entre une et vingt unités de temps.

[1] Le symbole | est utilisé pour représenter] ou [

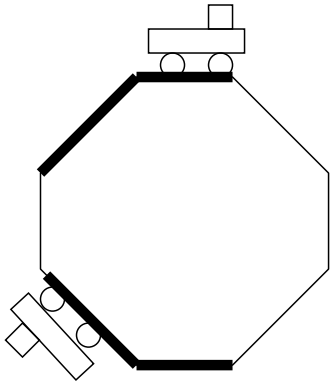


Figure VI.2: Exemple des trains de Genrich

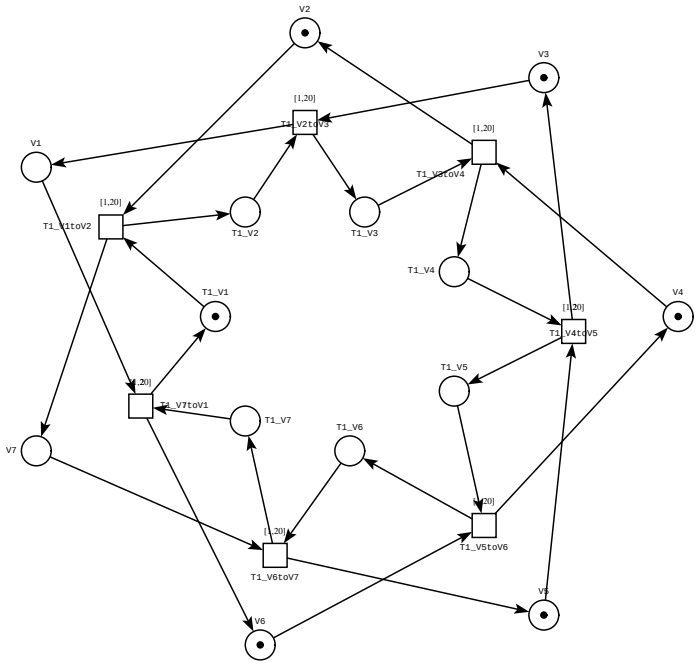


Figure VI.3: Réseau de Petri modélisant les trains

Cet exemple illustre l'explosion (à cause des contraintes temporelles) du graphe à explorer. Pour un système de 15 segments avec 3 trains on obtient un graphe de classes avec 128 704 classes et 337 608 transitions, tandis qu'une analyse atemporelle construit un graphe de 2 236 états avec 5 424 transitions.

D'après les résultats précédents pour étudier si ce système comporte des états de blocage on peut étudier le comportement atemporel. Puisque le comportement atemporel ne contient pas d'état de blocage, on peut en déduire directement que le comportement du système temporisé n'a pas d'état de blocage.

1.5 Extensions à des propriétés de LTL

Nous nous intéressons aux propriétés spécifiques et plus particulièrement aux propriétés exprimées par une formule LTL. Vérifier une propriété LTL sur un système consiste à vérifier que toutes les séquences d'exécution de ce système satisfont la propriété. Notons que vérifier une propriété LTL sur un système temporisé signifie que l'on tient compte des divergences temporelles. Un état est un état de divergence si toutes les transitions sensibilisées ont leur intervalle non supérieurement borné ou si aucune transition n'est sensibilisée.

Propriété 20 *Conservation des propriétés LTL*
Soit R un réseau de Petri, I_s et I'_s deux fonctions intervalles statiques telles que $\forall t, I_s(t) \subseteq I'_s(t)$. Soit Φ une propriété de LTL.

$$STE(R, I'_s) \models \Phi \Rightarrow STE(R, I_s) \models \Phi$$

Cette propriété est une implication directe de la conservation des traces maximales : s'il existe un contre-exemple maximal à la propriété dans le système initial, ce contre-exemple existe aussi dans la sur-approximation et est également un contre-exemple car il n'a pas pu être prolongé.

Remarquons que les propriétés de CTL ne sont pas obligatoirement conservées par une sur-approximation, puisque le branchement n'est pas obligatoirement respecté.

Notons également que toutes les méthodes de réductions classiques permettant de préserver les blocages ou des propriétés de LTL peuvent être combinées avec une méthode de sur-approximation. En particulier les techniques dites d'ordre partiel que nous avons exploitées dans les chapitres précédents.

1.6 Exemple de sur-approximation spécifique pour une propriété LTL.

Nous utilisons ici un exemple abordé dans le cadre du projet RNTL $COTRE$[2] et présenté dans [BRV+03b] : deux processus constitués de différentes tâches caractérisées par un temps minimal et un temps maximal s'exécutent en parallèle. Deux sémaphores permettent également de garantir l'exclusion mutuelle pour certaines opérations. Le code d'un processus est donné dans le langage $COTRE$ dans la figure VI.4. Le système est alors décrit par l'instanciation de deux processus et de deux sémaphores (cf figure VI.5). Les deux processus sont cycliques d'une période de 1396 et 1726 respectivement.

Une représentation de ce système par un réseau de Petri est donnée dans la figure VI.6. On observe les générateurs de période avec les transitions ui_T ($i \in \{1, 2\}$) qui permettent aux processus de recommencer leur exécution après avoir terminé l'exécution précédente. Dans le cas contraire la transition $deadline\_i$ signifie le dépassement du $deadline$ et arrête le processus.

On s'intéresse à la propriété ϕ suivante, qui exprime l'absence d'interblocage, en assurant que si un processus commence alors il termine obligatoirement.

$$\begin{aligned}\phi = \quad & [](u1\_Action1 \Rightarrow <> u1\_PeriodicWait) \\ & \wedge (u2\_Action1 \Rightarrow <> u2\_PeriodicWait)\end{aligned}$$

Pour vérifier cette propriété on souhaite construire une sur-approximation de ce réseau de Petri. On peut d'abord constater qu'une sur-approximation qui relâcherait les contraintes minimales des intervalles donnerait lieu à un graphe infini ; Effectivement les générateurs de période n'auront plus de contrainte minimale et par conséquent les places $u1\_Condition$ et $u2\_Condition$ ne seront plus bornées. Si on souhaite utiliser une sur-approximation en supprimant les contraintes maximales des intervalles, alors notre propriété ne sera trivialement plus vérifiée puisque tous

[2] COmposants Temps RÉel : http ://www.laas.fr/COTRE

```
component process(delta, period, deadline: thetatype;
          Pa, Pb, Va, Vb: out port)
  initially  PeriodicWait

  periodic(delta, period, deadline, prior)
  {
     t1: from  PeriodicWait -> periodic_wait;  Action1
  [] t2: from  Action1 -> delay(150,190);  WaitSemaphore1
  [] t3: from  WaitSemaphore1 when Pa! ->  Action2
  [] t4: from  Action2 -> delay(10,20);  WaitSemaphore2
  [] t5: from  WaitSemaphore2 when Pb! ->  Action3
  [] t6: from  Action3 -> delay(100,250);  ReleaseSemaphore1
  [] t7: from  ReleaseSemaphore1 when Va! ->  Action4
  [] t8: from  Action4 -> delay(100,150);  ReleaseSemaphore2
  [] t9: from  ReleaseSemaphore2 when Vb! ->  PeriodicWait
  }
end process
```

Figure VI.4: Code d'un processus en langage *COTRE*

```
connector P1, V1, P2, V2
subcomponent s1: semaphore(1, P1, V1)
subcomponent s2: semaphore(1, P2, V2)
subcomponent p1: process(0, 1396, 1396,
             P1, P2, V1, V2)
subcomponent p2: process(0, 1726, 1726,
             P1, P2, V1, V2)

```

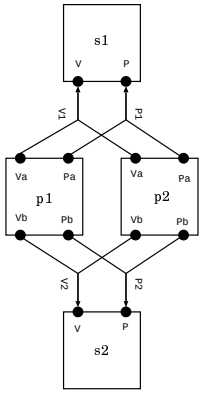


Figure VI.5: Instanciation du système en langage *COTRE*

les états des processus deviendront des états de divergence, par conséquent on n'aura plus la garantie que le processus termine.

La connaissance du système nous permet donc de définir une sur-approximation en relâchant uniquement les contraintes maximales du générateur de période : $[1396, 1396]$ devient $[1396, \infty[$ et $[1726, 1726]$ devient $[1726, \infty[$. Le graphe de classes (avec divergence) obtenu possède seulement 482 classes et satisfait la propriété ϕ. On peut donc conclure que le système initial satisfait la propriété ϕ sans avoir à construire son graphe de classes qui comporte 38 276 classes.

La sur-approximation permet également de vérifier des propriétés telles que l'exclusion mutuelle : $\phi_2 = []\neg(u1\_Action2 \wedge u2\_Action2)$.

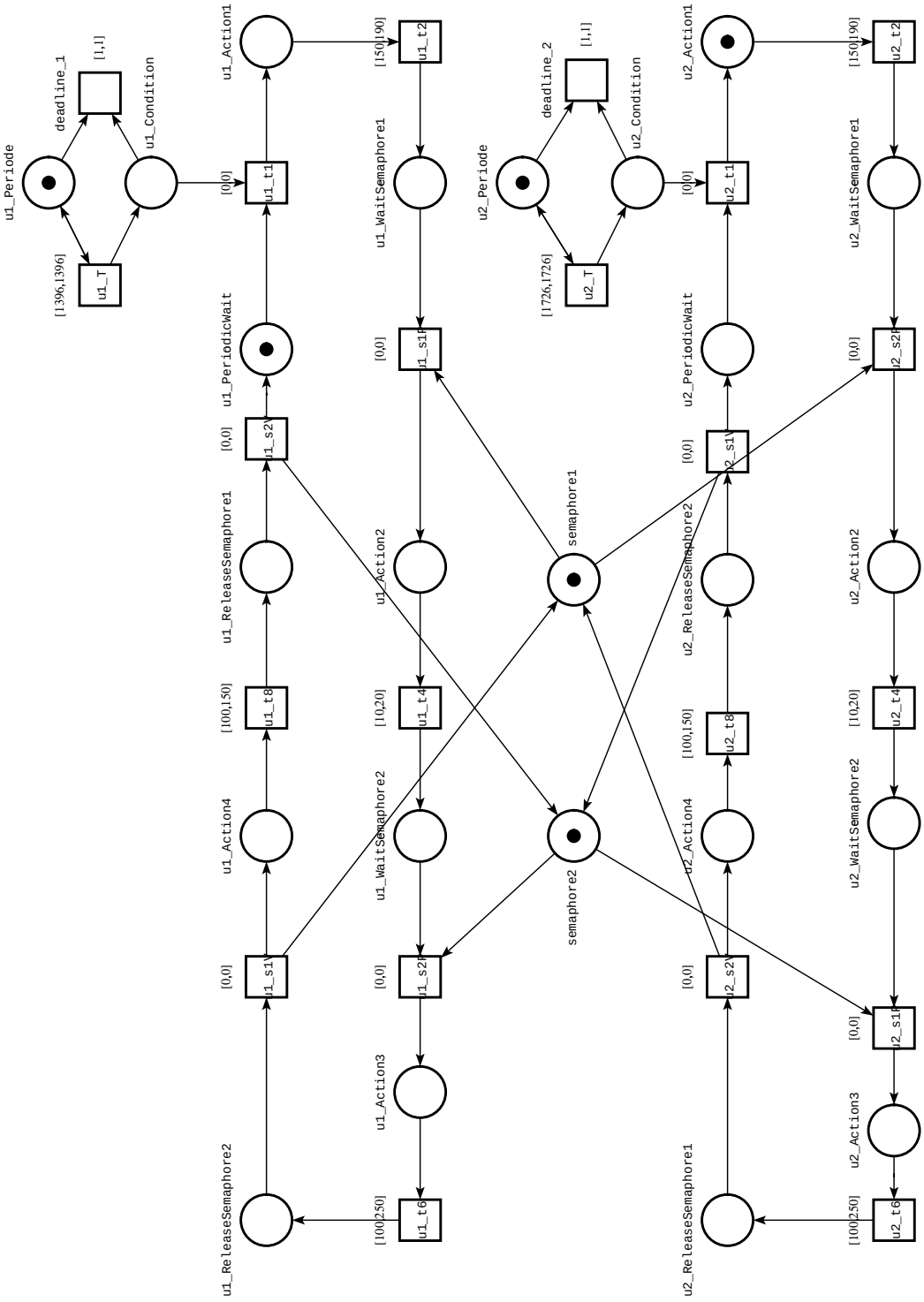


Figure VI.6: Modélisation en réseaux de Petri temporels de l'exemple « Deadlock »

2 Procédure de décision effective

L'approche par sur-approximation a cependant une limite majeure : dans le cas où la sur-approximation ne satisfait pas la propriété, on ne peut rien conclure. Pour pallier ce problème

notre objectif est d'utiliser les résultats obtenus sur la sur-approximation pour guider l'analyse du système.

Vérifier une formule LTL consiste à chercher s'il existe une séquence infinie qui soit un contre-exemple à la formule. Par conséquent si on a trouvé un contre-exemple, il faut vérifier si cette séquence est réalisable dans le système initial (avec les contraintes temporelles initiales).

Dans la pratique il faudrait donc chercher parmi toutes les séquences de la sur-approximation qui sont des contre-exemples si l'une d'elle est réalisable en respectant les contraintes temporelles. Même si le nombre de classes est fini, le nombre de ces séquences peut être infini. Pour prendre en compte toutes les séquences, nous considérons le graphe de toutes les classes apparaissant dans un contre-exemple. Pour analyser si les comportements de ce graphe sont temporellement réalisables, on construit alors la synchronisation de ce sous-graphe avec le graphe de classes. Le graphe synchronisé ne contient que les comportements qui sont temporellement réalisables. Le graphe synchronisé est alors analysé pour déterminer s'il contient effectivement des contre-exemples. On peut donc résumer les différentes étapes de l'approche par :

Phase 1 - Analyse de la sur-approximation

- Construire le graphe de classes de la sur-approximation.
- Rechercher dans ce graphe des contre-exemples à la formule.

S'il n'y a pas de contre-exemple, alors la propriété est vérifiée.

S'il y a au moins un contre-exemple, alors exécution de la seconde phase.

Phase 2 - Analyse du système guidée par les résultats

- « Elaguer » le graphe de classes de la sur-approximation : conserver uniquement les classes pouvant conduire à un état acceptant
- Construire le graphe synchronisé entre le graphe « élagué » et le graphe de classes.
- Rechercher une séquence contre-exemple dans le graphe synchronisé.

Nous allons détailler les étapes de la seconde phase permettant l'utilisation des résultats trouvés sur la sur-approximation pour guider la construction du graphe de classe du système. A partir du graphe de classe de la sur-approximation, on ne conserve que le sous-graphe induit par l'ensemble des classes pouvant conduire à un état acceptant pour l'automate de Büchi. Pour cela nous utilisons un algorithme de plus petit point fixe : l'algorithme est initialisé avec l'ensemble des classes acceptantes. A chaque itération, toutes les classes prédécesseurs d'une classe de l'ensemble est ajoutée. L'algorithme s'arrête lorsque l'ensemble n'augmente plus. Ce processus termine toujours puisque le graphe est fini.

La synchronisation permet alors de construire le graphe de classes du système guidé par ce graphe réduit : Une transition du graphe synchronisé est tirée uniquement si elle est tirable d'une part dans le graphe réduit et d'autre part si les contraintes temporelles du système le permettent.

La recherche d'un contre-exemple est alors effectuée dans le graphe synchronisé. Pour cela on cherche une séquence infinie dans le graphe contenant une infinité de fois un état acceptant, donc une composante fortement connexe contenant un tel état.

La figure VI.7 résume les différentes étapes de l'approche, que nous allons détailler dans la suite.

2.1 Construction du sous-graphe des contre-exemples

Comme nous l'avons vu, la formule n'est pas vérifiée lorsque l'automate synchronisé contient un état acceptant qui se trouve dans une composante fortement connexe contenant un cycle (une composante fortement connexe composée d'un seul état qui n'a pas de transition sur lui-même ne contient pas de cycle). Ces états jouent donc un rôle important et seront appelés « état concluant

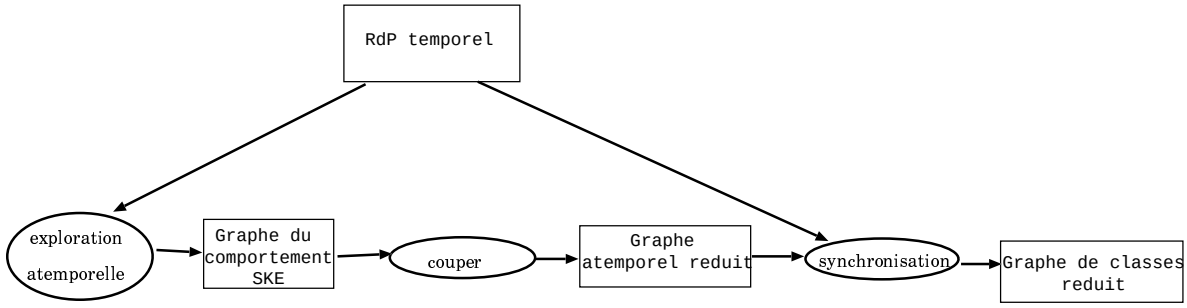


Figure VI.7: Vérification temporelle réduite par la vérification atemporelle

».

Pour construire le sous-graphe contenant tous les contre-exemples, nous considérons les états qui peuvent conduire à un état concluant.

Définition 67 *Sélection du sous-graphe des contre-exemples*
Soit $\Sigma = \langle S, s_0, T, \rightarrow \rangle$ le STE du système. On construit $\Sigma' = \langle S', s_0, T', \rightarrow' \rangle$ par :
- *Soit S' le plus petit ensemble fermé par la relation de précédence contenant les états concluants.*
- *La relation $\rightarrow'$ est définie par la restriction de la relation $\rightarrow$ à $S' \times S'$*
- *Enfin l'ensemble T' est défini par $T' = \{t \in T | \exists s_1, s_2 \in S' \text{ tel que } s_1 \stackrel{t}{\longrightarrow}' s_2\}$*

Cette définition dénotationnelle définit le sous-graphe choisi comme le sous-graphe induit par l'ensemble des états depuis lesquels un état concluant est accessible.

Pour implémenter la construction de ce graphe réduit nous utilisons un algorithme de plus petit point fixe : l'algorithme est initialisé avec l'ensemble des états concluant. Chaque itération ajoute à l'ensemble tous les états prédécesseurs de chaque élément de l'ensemble. Ce processus se termine lorsque l'ensemble n'augmente plus. Cette terminaison est garantie puisque nous travaillons sur un graphe fini ; on peut donc employer le terme d'algorithme.

La table VI.1 présente cet algorithme avec la fonction $Pre(s) = \{s_1 \in S | \exists t \in T \text{ et } (s_1, t, s) \in G\}$. Toutes les transitions reliant deux sommets sélectionnés sont considérées dans le sous-graphe induit, il est donc inutile d'enregistrer les transitions : l'ensemble des états sélectionnés est suffisant.

Soit S et G donnés Et C l'ensemble des états concluants

$S' \leftarrow C$ /* Ensemble des états conservés */
Loop
  $Q \leftarrow \emptyset$
  For s **in** S' **do**
  $Q \leftarrow Q \cup (Pre(s) \setminus S')$
  end for ;
  $S' \leftarrow S' \cup Q$
Until $Q = \emptyset$

Table VI.1: Algorithme de sélection du sous-graphe

2.2 Synchronisation du graphe sélectionné et du graphe de classes

L'étape suivante consiste à vérifier si l'un des contre-exemples trouvés sur la sur-approximation est réalisable dans le système réel. Pour cela on synchronise le graphe des contre-exemples de la sur-approximation avec le graphe de classes du système initial.

Définition 68 *Synchronisation entre le Graphe de Classe et le graphe des contre-exemples*
Soit $\Sigma_A = \langle S_A, s0_A, T_a, \rightarrow_A \rangle$ *le STE des contre-exemples et* $\Sigma_T = \langle S_T, s0_T, T_A, \rightarrow_T \rangle$ *le graphe de classes. Le graphe synchronisé* $\Sigma' = \langle S', s0', T', \rightarrow' \rangle$ *est défini par :*

- $S' = S_A \times S_T$
- $s0' = (s0_A, s0_T)$
- $(s1, t, s2) \in \rightarrow'$ *ssi* $(s1_A, t, s2_A) \in \rightarrow_A$ *et* $(s1_T, t, s2_T) \in \rightarrow_T$ *avec* $s1 = (s1_A, s1_T)$ *et* $s2 = (s2_A, s2_T)$

En pratique l'objectif est de ne pas construire le graphe de classes du système initial. Par conséquent il n'est pas construit comme préliminaire à la synchronisation, mais on construit le graphe synchronisé en explorant au vol les parties du graphe de classes concernées. L'algorithme est donné dans la table VI.2.

Soit S' l'ensemble des classes sélectionnées

```
Pile ← s0    /*s0 = (m0, I0) */
S ← {s0}
G ← ∅ /* Graphe calculé */
while NotEmpty(Pile) do
    s = (m, I) ← pop(Pile)
    if Enabled(m) = ∅ then
        print "Blocage Effectif"
        exit
    endif
    E ← Enabled(m)
    for each t in E do
        m' ← fire(m, t)
        if m' ∈ S' then /* Synchronisation */
            s' ← fire(s, t) /* Calcul de la classe atteinte*/
            G ← G ∪ {⟨s, t, s'⟩}
            if s' ∉ S then
                S ← S ∪ {s'}
                push(Pile, s')
            endif
        endif
    endfor
endwhile
```

Table VI.2: Algorithme de calcul du graphe de classes synchronisé

2.3 Exemple d'analyse guidée par les résultats d'une sur-approximation

Nous proposons une variante des trains de Genrich[Gen87] temporisés présentés précédemment : initialement un seul train se trouve sur la voie. Ce train a un problème de fonctionnement,

soit il peut tomber définitivement en panne et s'arrêter sur la voie (plus aucun train ne peut alors circuler), soit le machiniste arrive à le réparer, auquel cas le train retrouve une marche normale et d'autres trains peuvent être ajoutés sur la voie.

L'objectif est de vérifier si ce système comporte ou pas des états de blocage.

Pour un modèle constitué de 30 segments et 3 trains, le graphe de classes [BRV04] ne peut pas être construit sur une station Sparc[3]. Nous ne pouvons donc pas vérifier si ce système comporte des états de blocage. Nous proposons d'utiliser la sur-approximation qui relâche les contraintes minimales et maximales. Le graphe d'accessibilité de cette sur-approximation contient seulement 22 053 états et peut donc être construit. Malheureusement ce graphe comporte un état de blocage et ne permet donc pas de conclure si le système comporte effectivement un blocage ou pas. Il est donc nécessaire d'utiliser notre approche de guidage pour vérifier si ce blocage est accessible dans le système initial. La réduction de ce graphe à la partie pouvant conduire au blocage permet de ne conserver un sous-graphe avec seulement 32 états. Finalement le graphe de classes synchronisé avec ce sous-graphe comporte uniquement 32 classes avec un blocage. On peut donc conclure que le système comporte bien un état de blocage et donner une séquence conduisant à ce blocage.

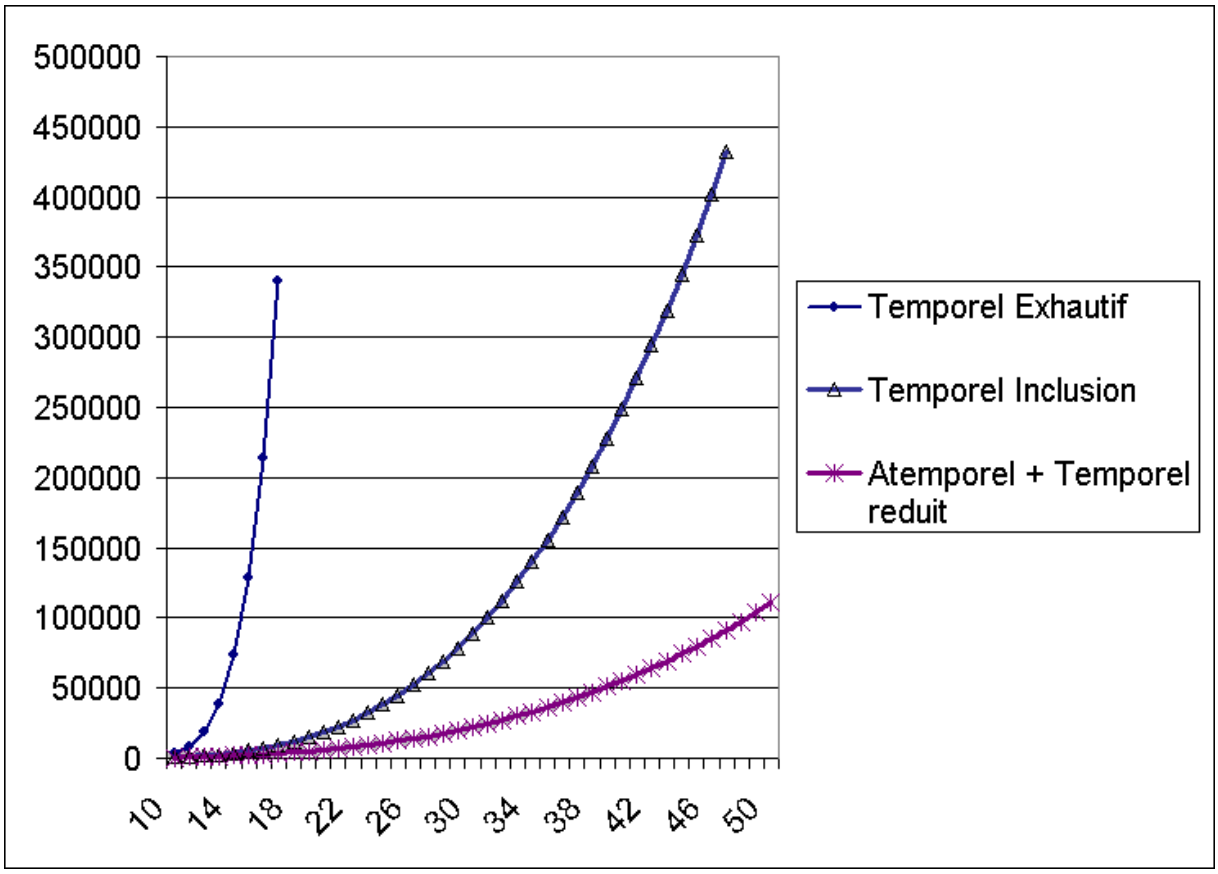


Figure VI.8: Taille du graphe pour 3 trains en fonction du nombre de segments

La figure VI.8 représente la taille des différents graphes construits. La courbe « Temporel exhaustif » représente le graphe de classe classique. La courbe « Temporel Inclusion » utilise l'algorithme permettant de réduire le nombre de classes en utilisant non plus l'égalité des classes

[3]ultra-5 avec 2 Gigas octets de mémoire

construites mais l'inclusion[4]. Enfin la courbe « Atemporel + temporel réduit » correspond à la somme de la taille du graphe construit par la sur-approximation et du graphe temporel synchronisé construits par notre approche. On observe donc que la méthode a permis ici de réduire considérablement l'explosion du nombre de classes construites.

En modifiant les temporisations pour que le machiniste arrive toujours à réparer le train, l'état de blocage devient temporellement inaccessible. Pour 3 trains avec 20 segments, le graphe atemporel exhaustif a 5903 états dont seulement 22 états peuvent conduire à l'état de blocage. Le graphe de classes synchronisé contient seulement 22 classes, sans état de blocage, tandis que le graphe de classes total qu'il aurait fallu construire sans notre méthode contient 1 099 603 classes (et 3 038 918 transitions).

2.4 Études des limites du guidage

Pour certains systèmes il est très difficile de guider efficacement l'analyse par les résultats de la sur-approximation. Cette situation se caractérise lorsque la sélection du sous-graphe de la sur-approximation ne permet pas de réduire sa taille, en effet cela signifie que l'on ne pourra pas réduire la taille du graphe de classes du système.

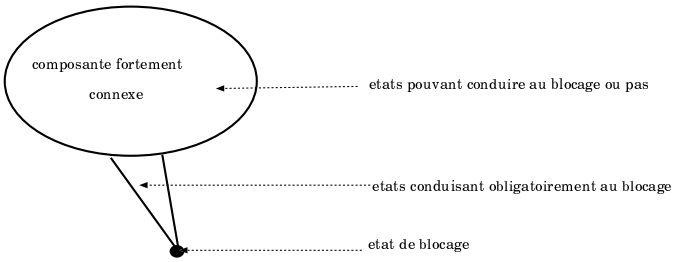


Figure VI.9: Comportement dont l'analyse atemporelle ne permet pas de guider l'analyse temporelle

La figure VI.9 symbolise un exemple de comportement de la sur-approximation qui ne permet pas de guider. Le comportement de la sur-approximation est constitué d'une composante fortement connexe et d'une partie conduisant obligatoirement à un état de blocage. Donc si on cherche à savoir si ce blocage de la sur-approximation est temporellement accessible dans le système initial, on s'intéresse à tous les états antécédents de cet état de blocage, donc au graphe entier.

L'absence de réduction s'explique par le fait que dans le système initial, il est possible que le blocage ne soit pas accessible par une séquence acyclique, mais uniquement après une longue période de fonctionnement durant laquelle un décalage progressif entre différentes horloges va se produire, par conséquent l'analyse temporelle doit tenir compte de ces comportements pour vérifier si l'état de blocage est accessible ou pas. Par exemple dans le réseau de Petri de la figure VI.10, l'état de blocage $p4$ n'est pas directement accessible par la séquence de transition $t0.t2.t4$: il faut 3 tours de la boucle de gauche et 2 tours de la boucle de droite avant que la transition $t4$ soit tirable. Le graphe de classes obtenu (représenté figure VI.11) met en évidence que la classe atteinte après « un tour » du système de gauche et « un tour » du système de droite est différente de la classe initiale.

[4]Si une classe c a le même marquage qu'une classe c' avec le domaine temporel de c inclus dans le domaine temporel de c', alors on remplace c par c'

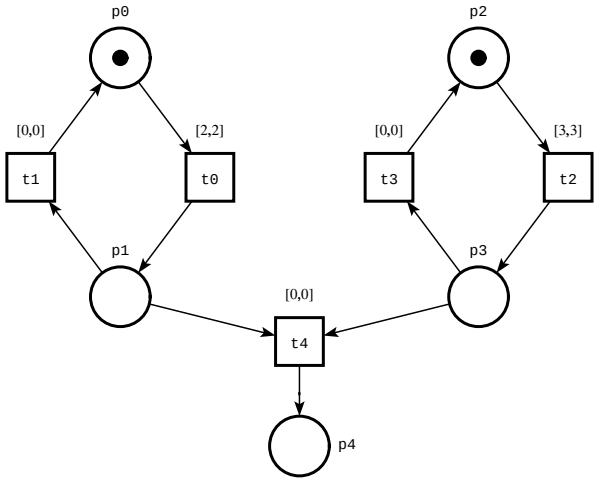


Figure VI.10: Exemple nécessitant plusieurs boucles

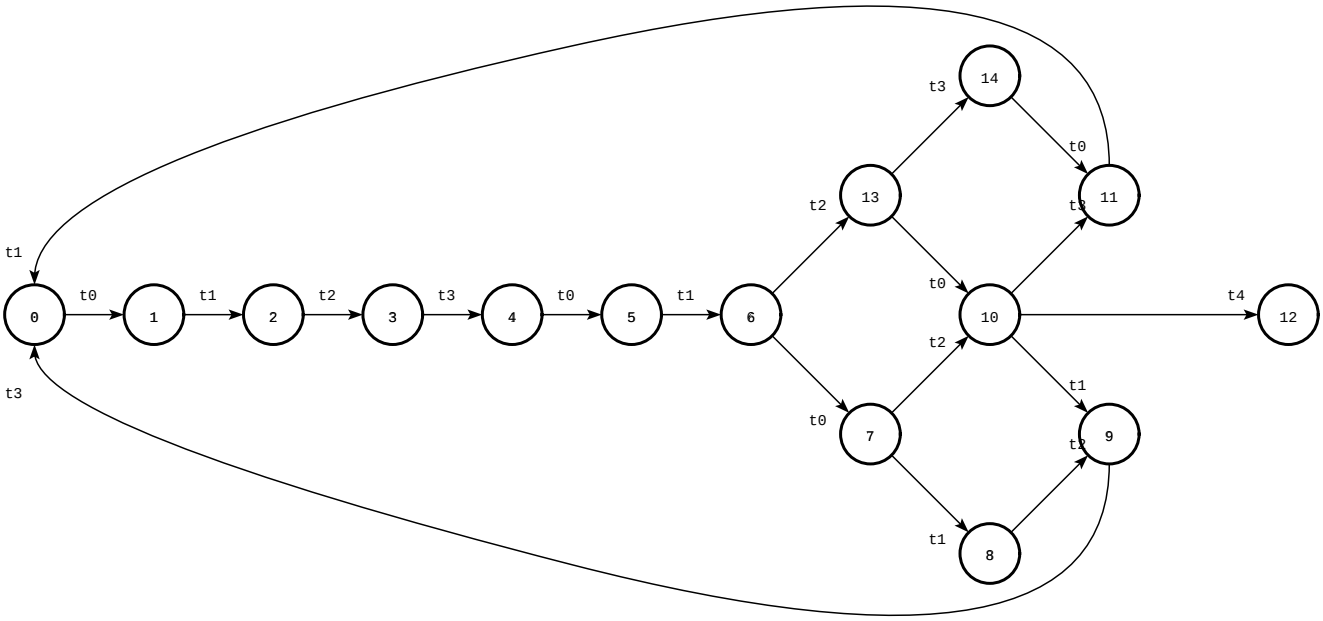


Figure VI.11: Graphe de classes

Inversement la figure VI.12 représente un comportement d'une sur-approximation qui permettra probablement un guidage efficace de l'analyse du système, puisque la sélection des antécédents de l'état de blocage permet de sélectionner un sous-ensemble réduit.

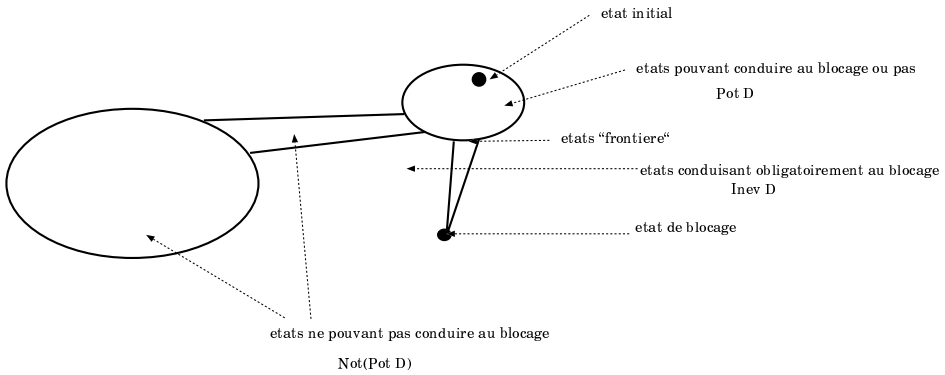


Figure VI.12: Comportement où l'approche par sur-approximation peut être efficace

2.5 Vérification par sur-approximation et par bandes

2.5.1 Présentation de l'approche par bandes

Dans l'approche proposée à la section précédente, la vérification d'une propriété LTL consiste à chercher tous les contre-exemples de la formule sur la sur-approximation et vérifier si l'un d'eux est réalisable en respectant les contraintes temporelles du système initial. Cette approche explore en une seule fois tous les antécédents des états concluants. Pour un système complexe, le sous-graphe ainsi construit peut avoir une taille encore importante. Nous proposons donc une variante à cette approche qui consiste à faire une vérification en plusieurs morceaux ou « bandes ».

Intuitivement chaque contre-exemple de la propriété trouvé sur la sur-approximation, va être analysé pour vérifier si il est réalisable dans le système originel. Dès que l'un d'eux est réalisable dans le système, on peut affirmer que la propriété n'est pas vérifiée.

Ainsi, chaque composante fortement connexe contenant au moins un état concluant est considérée séparément pour analyser si un état concluant est accessible dans le système initial : Pour une composante fortement connexe contenant un état concluant on considère le sous-graphe de tous les états permettant d'atteindre cette composante. Ce sous-graphe est alors synchronisé avec le graphe de classes du système initial et analysé. Ces opérations sont refaites pour chaque composante fortement connexe. Des parties du graphe seront éventuellement construites plusieurs fois puisque les « bandes » successivement construites forment un recouvrement du graphe construit par l'approche de la section précédente.

2.5.2 Exemple de vérification par bandes

Nous allons montrer un exemple de vérification par bandes pour la vérification des états de blocage.

Le système est constitué par n processus identiques que nous nommerons *Cycleurs* qui passent cycliquement dans les états a_i, b_i, c_i, d_i et retournent ensuite à leur état initial a_i. Les changements d'états a_i vers b_i et b_i vers c_i ne dépendent que de l'état du cycleur i. Avant de pouvoir se réinitialiser, un cycleur doit se synchroniser avec l'un de ses homologues (passage de l'état c_i à d_i) et ensuite refaire le plein d'énergie (d_i vers a_i). Le Figure VI.13 présente un processus cycleur, avec deux transitions de synchronisation (synchronisation avec le cycleur 0 ou avec le cycleur 2), et trois transitions permettant de revenir dans l'état initial en reprenant de l'énergie en fonction de l'état du réservoir.

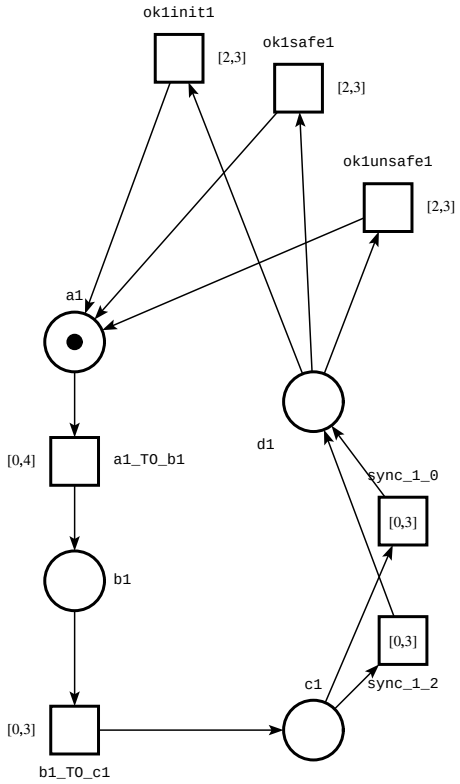


Figure VI.13: Un cycleur

Le réservoir d'énergie est modélisé par un processus spécifique que nous nommerons *réservoir*. Dans la modélisation proposée, tous les cycleurs se synchronisent avec le même processus *réservoir*. Le comportement du processus *réservoir* est le suivant : initialement, il est apte à délivrer de l'énergie.

- La transition *tostop*1 le place dans un état où il ne délivrera plus jamais d'énergie : une fois que cette transition est franchie le blocage global du système est irrémédiable.
- La transition $tosafe$ le fait évoluer vers un état où il sera toujours disposé à fournir de l'énergie (celle-ci ne viendra donc jamais à manquer) et le système sera exempt de blocage. Mais cette transition n'est exécutable qu'après la transition *latence* donc après 10 à 50 unités.
- La transition $tounsafe$ le place dans un état où il est toujours capable de délivrer de l'énergie (le blocage n'est pas inévitable) mais il ne pourra plus jamais tirer la transition $tosafe$ même après la *latence*. Il peut alors franchir la transition *tostop*2 qui le conduit dans un état où il ne délivre plus jamais d'énergie : le blocage global du système est donc irrémédiable.

La figure VI.14 représente le réseau de Petri du processus réservoir, qui délivre de l'énergie aux cycleurs lorsque une des places $Einit$, $Esafe$ ou $Eunsafe$ est marquée.

La figure VI.15 représente deux processus cycleurs ainsi que le processus réservoir. Pour simplifier le dessin les différentes transitions de synchronisation entre les cycleurs et le réservoir ont été superposées.

Le comportement atemporel d'un système à deux cycleurs comporte 144 états et 378 transitions. Ces différents états peuvent être regroupés en fonction de leur caractéristique par rapport aux états de blocages. On peut en particulier s'intéresser aux propriétés suivantes :

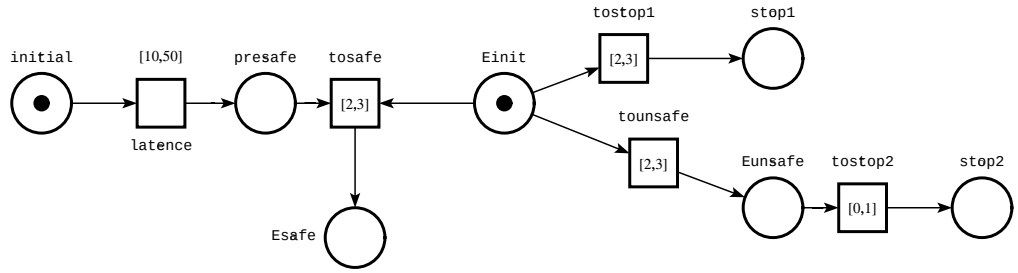


Figure VI.14: Processus réservoir

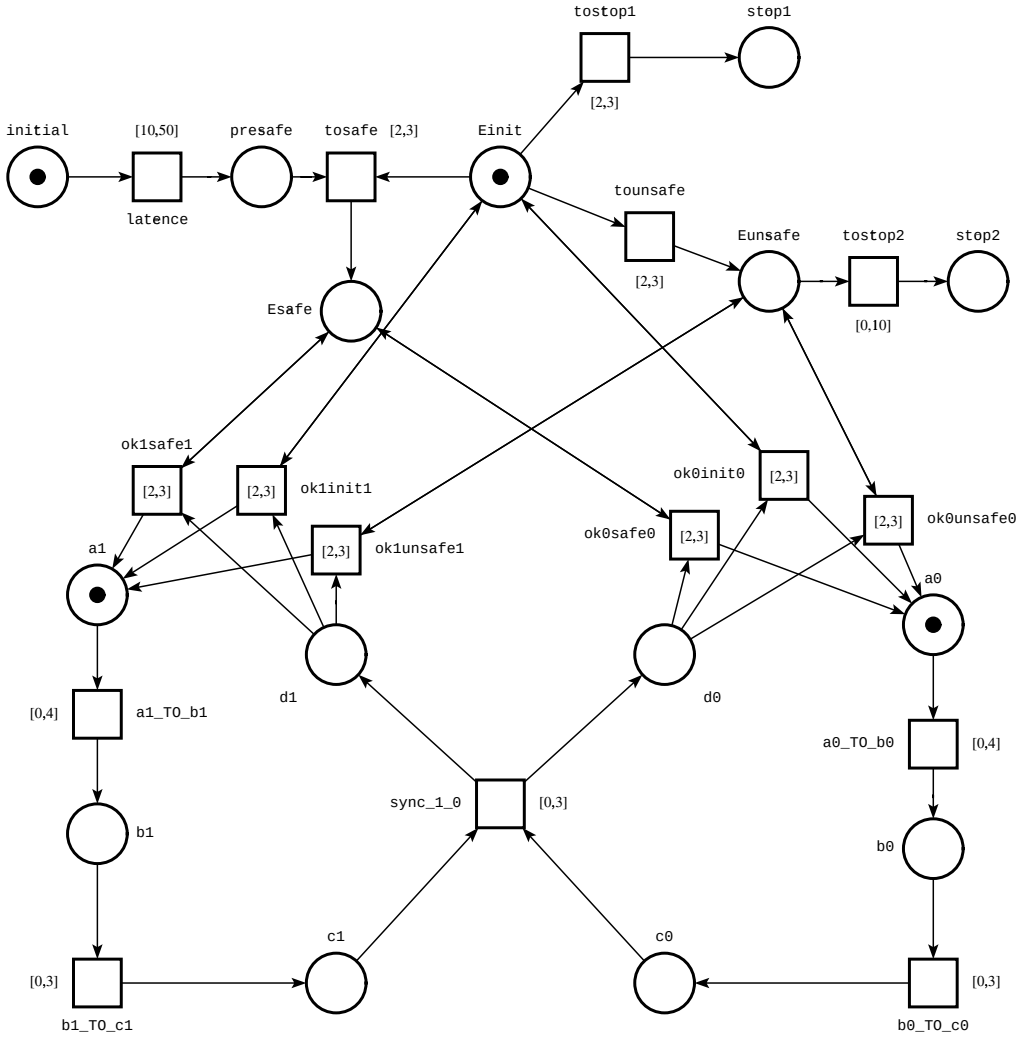


Figure VI.15: Système de deux cycleurs et du processus réservoir

– le blocage n'est pas accessible (Zone 1)
– le blocage est accessible et on peut aussi aller dans un état où le blocage n'est pas accessible (Zone 2)
– le blocage est accessible et le restera, mais n'est pas inévitable (Zone 3)

– le blocage est inévitable (Zone 4)

Ces propriétés présentent l'intérêt de partitionner l'ensemble des états du système. Quel que soit le système considéré, les transitions permettant de passer d'un état d'une zone à un état d'une autre zone sont spécifiques. La figure VI.16 représente les différents liens qui peuvent exister entre ces différents ensembles. Suivant les caractéristiques du système considéré ces différents ensembles pourront avoir des tailles plus ou moins importantes, ou même être complètement vide. De même l'état initial du système peut se trouver dans n'importe lequel de ces ensembles. Une autre caractéristique intéressante sera le nombre d'états frontières : par exemple ce schéma montre que certains états de la zone 2 peuvent avoir une transition permettant d'aller dans un état de la zone 1 et une transition allant dans un état qui reste dans la zone 2, donc suivant le choix fait dans cet état frontière on détermine le futur du système. Le nombre et la liste de ces états peuvent être utiles pour appréhender le comportement du système.

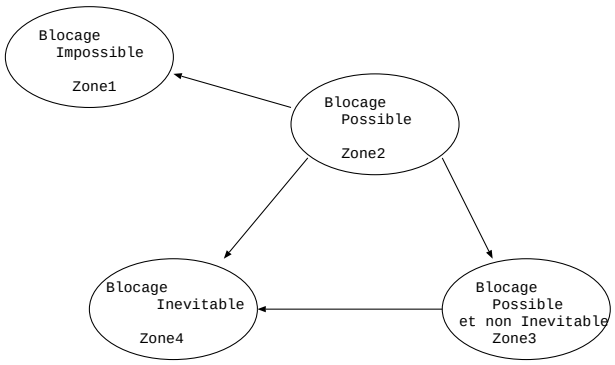


Figure VI.16: Partitionnement des états d'un système

Dans l'exemple des cycleurs et du processus réservoir considéré précédemment, le partitionnement des états est déterminé par l'état du processus réservoir. Initialement tout est possible (blocage accessible mais pas obligatoirement, etc...), par conséquent l'état initial, caractérisé par le marquage de la place $Einit$, se trouve dans la zone 2. Le tir de la transition $tosafe$ fait passer le réservoir dans l'état $Esafe$, le système se retrouve alors dans la zone 1 et n'en sortira plus. La transition $tostop1$ fait passer directement dans la zone 4, tandis que la transition $tounsafe$ fait passer dans la zone 3. A partir de la zone 3, le tir de la transition $tostop2$ rend le blocage inévitable et conduit donc dans la zone 4. En fait si l'on regarde plus précisément l'état de blocage atteint n'est pas le même, on peut donc raffiner le schéma précédant en distinguant suivant le blocage atteint : la figure VI.17 illustre le schéma obtenu pour l'exemple du réservoir. Suivant les temporisations choisies, certains comportements ne seront pas possibles.

Avec les temporisations de la Figure VI.15 le graphe de classes comporte 2 354 classes pour 2 cycleurs alors que le comportement atemporel avait seulement 144 états. Une sur-approximation du comportement peut donc être intéressante. La table VI.3 donne la taille du graphe de classes ainsi que la taille de la plus grande et de la plus petite bande construite. Cette approche permet donc de trouver des états de blocage au système en ayant construit un graphe beaucoup plus petit.

Avec des temporisations différentes (par exemple une contrainte $[2, 3]$ pour les transitions des cycleurs de l'états b à l'état c) le blocage devient irrémédiable. L'état initial du système se trouve donc cette fois dans la zone 4. La table VI.4 donne les résultats obtenus pour cette nouvelle configuration.

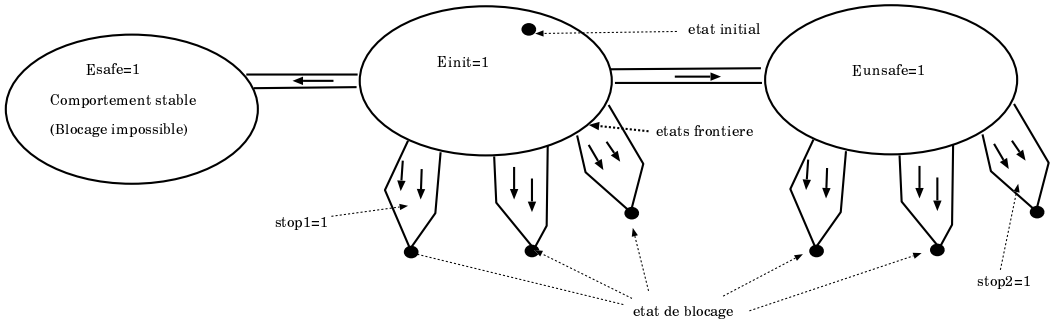


Figure VI.17: Les cycleurs - Partitionnement des états

Taille modèle	Graphe de classes	Plus grande bande	Plus petite bande
2 cycleurs	2 354	1 737	520
3 cycleurs	33 355	25 733	12 523
4 cycleurs	401 068	324 311	175 173

Table VI.3: Les cycleurs - Vérification par bandes

Taille modèle	Graphe de classes	Plus grande bande	Plus petite bande
2 cycleurs	173	119	15
3 cycleurs	1 664	1 078	107
4 cycleurs	20 020	14 510	916
5 cycleurs	293 271	198 886	10 942

Table VI.4: Les cycleurs avec blocage irrémédiable - Vérification par bandes

2.6 Utilisation des ordres-partiels pour le guidage

Une sur-approximation particulière est la sur-approximation qui relâche les contraintes minimales et maximales. Cette sur-approximation correspond à une analyse atemporelle du système en tenant compte des états de divergence. Comme nous l'avons vu un état de divergence est très simple à détecter puisque il dépend uniquement des intervalles statiques. L'analyse de cette sur-approximation peut donc être faite avec les techniques de réduction que nous connaissons pour les systèmes atemporels à condition que la technique conserve la propriété que l'on cherche à démontrer.

Dans cette section nous montrons comment les techniques de réduction basées sur les ordres partiels peuvent être utilisées pour la vérification des états de blocage. Cette approche permet donc d'une part de limiter l'explosion due aux contraintes temporelles par l'utilisation de la sur-approximation et d'autre part de réduire l'explosion due à la représentation du parallélisme par l'entrelacement d'actions en utilisant des techniques de réduction ordres-partiels.

2.6.1 Utilisation des graphes de pas couvrants

La technique des graphes de pas couvrants permet de construire une représentation abstraite du comportement atemporel du système plus petite que l'exploration exhaustive. Cette abstraction préserve tous les états de blocage du système. L'idée est donc d'utiliser les graphes de pas couvrants pour déterminer tous les états de blocage atemporel du système tout en construisant ainsi un graphe atemporel de taille réduite.

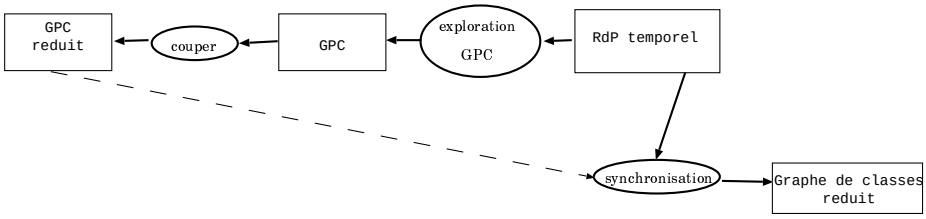


Figure VI.18: Utilisation des Graphes de Pas couvrants pour une vérification temporelle : solution idéale

Si l'exploration atemporelle par un graphe de pas couvrants ne contient pas de blocage alors on peut en conclure que le système initial n'a pas d'état de blocage.

Dans le cas où le graphe de pas couvrants construit contient des états de blocage il faut vérifier si ces états sont accessibles par le système initial. Idéalement on voudrait analyser un système en construisant le graphe de pas couvrants atemporel et utiliser ce graphe pour « guider » l'exploration temporelle, comme présenté Figure VI.18.

Cependant pour savoir si un état de blocage du graphe atemporel est accessible par le système temporisé, il faut analyser tous les chemins qui conduisent à l'état de blocage. En effet un chemin conduisant à l'état de blocage peut être temporellement impossible, tandis qu'un autre chemin conduisant au même blocage peut être le support d'un échéancier. Les graphes de pas couvrants conservent tous les états de blocage du système, malheureusement ils ne permettent pas de récupérer simplement tous les chemins conduisant à ces états de blocage. Par conséquent il n'est pas trivial de guider l'exploration temporelle par le graphe de pas couvrants (d'où la représentation en pointillés sur la Figure VI.18 de la flèche entre GPC réduit et synchronisation).

2.6.2 Récupération de toutes les séquences conduisant aux blocages

Le graphe de pas couvrants fournit tous les états de blocages accessibles, cependant toutes les séquences permettant de les atteindre ne sont pas conservées. Il est donc nécessaire de trouver une technique permettant de construire toutes les séquences atemporelles conduisant à un état de blocage. Une première technique permettant d'atteindre ce but est d'abord proposée. Cette technique sera alors améliorée dans la section suivante.

On ne conserve tout d'abord qu'un sous-graphe du GPC : on considère le sous-graphe induit par les états qui peuvent conduire à un état de blocage. Les transitions du système n'apparaissant pas dans ce sous-graphe ne peuvent pas conduire à des états de blocage. Par la suite on ne va donc tenir compte que des transitions présentes dans ce sous-graphe.

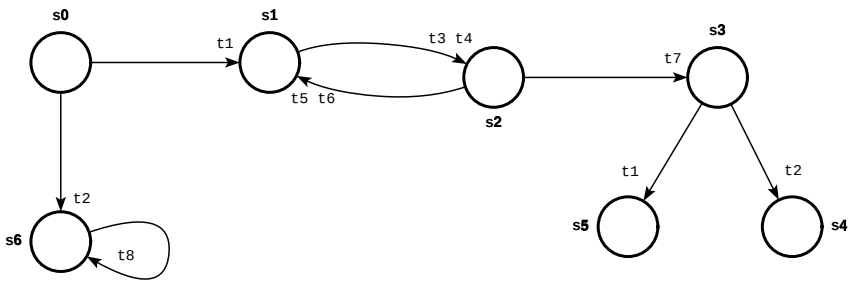


Figure VI.19: Graphe de pas couvrants complet

Pour exemple considérons le GPC de la figure VI.19. Les états de blocage sont les états $s4$ et $s5$. Par conséquent la sélection du sous-graphe permettant d'atteindre les blocages donne le graphe de la figure VI.20. Nous en déduisons que seules les transitions $\{t1, t2, t3, t4, t5, t6, t7\}$ (Pas la transition $t8$) permettent éventuellement d'atteindre un blocage.

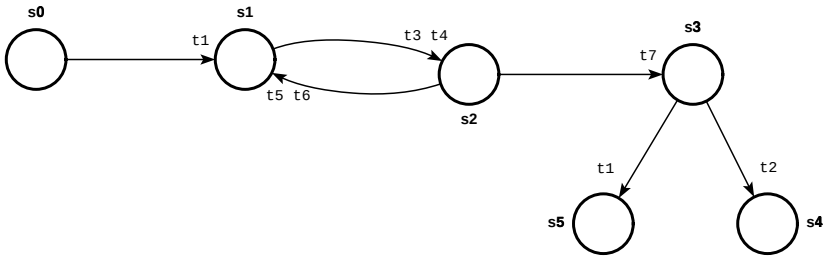


Figure VI.20: Graphe de pas couvrants sélectionné

On utilise alors cet ensemble de transitions pour construire la partie du graphe exhaustif atemporel qui contient potentiellement des blocages ; pour cela on fait une exploration atemporelle classique en n'autorisant que le tir de transitions présentes dans cet ensemble. Par exemple dans le cas précédent, on ne s'autorise à tirer que les transitions $t1$,$t2$,$t3$,$t4$,$t5$,$t6$ et $t7$.

Notons que certains états du graphe construit n'auront aucun successeur car aucune transition de l'ensemble ne sera sensibilisée, pour autant l'état n'est pas un état de blocage du système car une transition extérieure à l'ensemble peut être tirée.

Tous les états du graphe construit ne permettent pas d'atteindre un état de blocage. On utilise donc à nouveau la sélection d'un sous-graphe par plus petit point fixe en initialisant avec les états de blocage et en prenant les antécédents à chaque itération (comme dans le cas général) : le sous-graphe obtenu ne contient que des états qui peuvent conduirent à des états de blocage. Ensuite, comme dans la démarche précédente, on construit le graphe temporel synchronisé avec ce sous-graphe atemporel. La figure VI.21 résume les différentes étapes de la démarche.

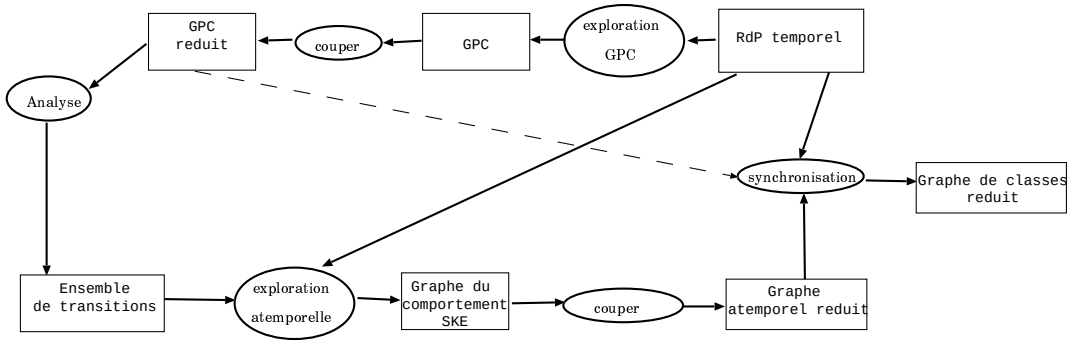


Figure VI.21: Utilisation des Graphes de Pas couvrants pour une vérification temporelle

Remarquons que le graphe atemporel réduit utilisé au final pour la synchronisation avec le graphe de classes est identique au graphe construit sans l'utilisation des pas couvrants : en effet dans les deux cas il correspond à tous les comportements atemporels pouvant conduire à un blocage. L'intérêt de l'utilisation des pas consiste à éviter la construction de l'atemporel exhaustif. La première partie de cette thèse a montré la réduction qui pouvait être obtenue par l'utilisation d'un graphe de pas couvrants.

2.6.3 Affinement de la méthode

Dans la méthode précédente, après avoir choisi un sous-graphe du GPC, toutes les transitions apparaissant dans ce graphe sont utilisées autant de fois que possible pour la construction du graphe atemporel. Nous souhaitons affiner la construction de cet ensemble de transitions « utiles » pour réduire la taille du graphe atemporel exploré. On ne va donc plus construire un ensemble de transitions que l'on s'autorise à tirer autant de fois que l'on veut, mais un « multi-ensemble » de transitions ; à chaque transition est associé le nombre maximal d'apparitions permettant d'arriver à un blocage. En fait les transitions présentes dans un cycle doivent être permises autant de fois qu'on le souhaite, par conséquent on utilise un multi-ensemble généralisé qui permet l'utilisation de ω pour ces transitions.

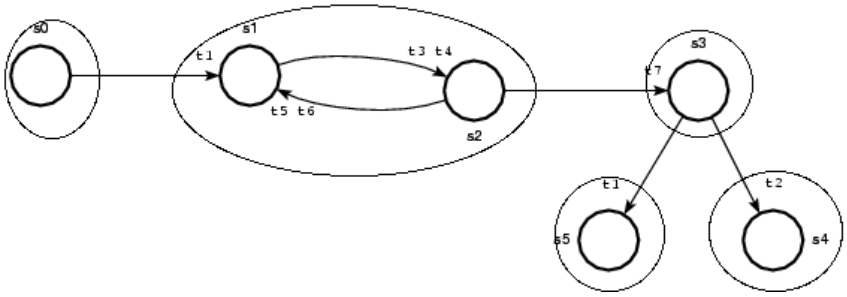


Figure VI.22: Composantes connexes du graphe de pas couvrants sélectionné

Pour déterminer le nombre maximal de tirs utiles d'une transition pour atteindre un état de blocage, nous allons considérer les composantes fortement connexes. En effet les transitions présentes dans une composante fortement connexe peuvent être tirées autant de fois que l'on veut avant d'atteindre le blocage. On commence donc par construire sur le GPC, les composantes fortement connexes. Lorsqu'un pas relie deux composantes fortement connexes, on ajoute chaque transition du pas au multi-ensemble de transitions. Dans l'exemple de la figure VI.22, c'est le cas des transitions t_1,t_2 et t_7. La transition t_1 apparaît deux fois, donc elle sera ajoutée deux fois dans le multi-ensemble. Dans le cas où la composante connexe contient plusieurs pas alors chaque transition apparaissant dans un pas de la composante fortement connexe est mise dans le multi-ensemble de transition avec une puissance ω pour signifier qu'on ne limite pas le nombre de fois qu'elle peut être tirée. Sur notre exemple, la composante fortement connexe composée des états $s1$ et $s2$ provoque l'ajout des transitions des pas $\{t3; t4\}$ et $\{t5; t6\}$ avec des puissances ω. Finalement le multi-ensemble calculé pour notre exemple est : $\{t_1^2, t_2, t_3^\omega, t_4^\omega, t_5^\omega, t_6^\omega, t_7\}$

L'exploration atemporelle est alors menée avec ce nouveau multi-ensemble, ce qui signifie que dans chaque état construit lors de l'exploration atemporelle, il faut maintenir un multi-ensemble correspondant aux transitions « autorisées ». Dans l'état initial, les transitions autorisées sont exactement le multi-ensemble calculé grâce au GPC. Lors du tir d'une transition, on calcule le nouveau multi-ensemble de l'état d'arrivée en retirant une occurrence de cette transition du multi-ensemble[5]. Deux séquences peuvent arriver dans le même marquage en ayant utilisé les mêmes transitions dans un ordre différent, dans ce cas, les deux séquences arrivent dans le même état. Dans le cas où les transitions utilisées, ne sont pas identiques, il se peut que l'on ne puisse pas comparer les deux multi-ensembles, puisque l'inclusion n'est qu'un ordre partiel, par conséquent les deux états atteints seront différents (mais avec des marquages identiques). Au final, le graphe obtenu (qui n'est pas un DAG, car les puissances infinies dans le multi-

[5] une puissance ω reste identique lors de la soustraction d'une unité !

ensemble, peuvent permettre des boucles) peut avoir plusieurs états avec le même marquage, mais comme nous l'avons vu nous avons uniquement besoin de la liste des états accessibles pour la synchronisation, donc il est inutile de quotienter le graphe obtenu par rapport aux marquages. On ne conserve que la liste des marquages accessibles pour faire la synchronisation comme dans l'approche initiale. La figure VI.23 récapitule les différentes étapes de cette approche, mais cette solution n'a pas encore été implémentée.

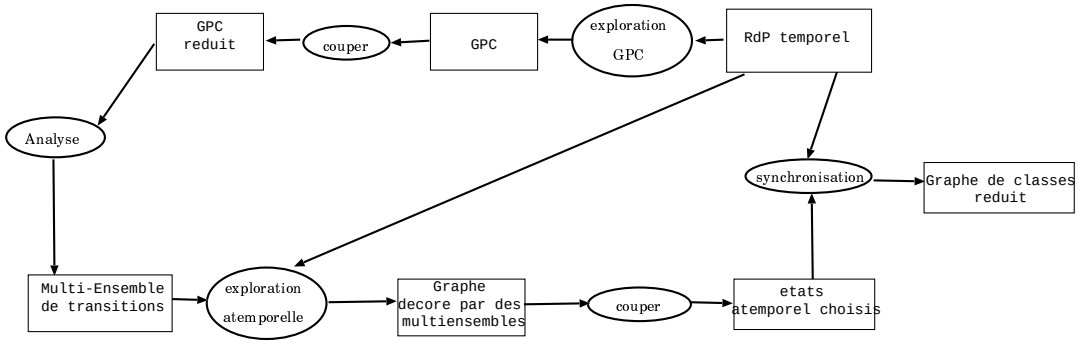


Figure VI.23: Utilisation des Graphes de Pas couvrants pour une vérification temporelle

3 Conclusion

Partant de l'observation, que dans certains formalismes (dont les réseaux de Petri temporels) les contraintes temporelles n'ajoutent pas d'état de blocage temporel (blocage dû aux contraintes temporelles), nous avons montré que relâcher des contraintes temporelles permet de construire une sur-approximation du comportement. Plus précisément, le comportement temporellement contraint simule le comportement relâché, ce qui nous permet de garantir la préservation des états et des séquences. Par conséquent des propriétés de blocage ou des propriétés LTL peuvent être vérifiées sur la sur-approximation.

Dans le cas où l'analyse de la sur-approximation ne permet pas de conclure, nous avons défini une seconde étape d'analyse du système par une exploration guidée par les résultats obtenus sur la sur-approximation.

Un prototype, réalisé à partir du noyau de l'outil *Tina*, a déjà permis de montrer qu'on obtenait des performances intéressantes sur certains exemples classiques.

Conclusion

La part de l'informatique dans les systèmes est toujours croissante. De plus les systèmes remplissent des fonctions de plus en plus critiques. On observe donc d'une part une augmentation de la difficulté à assurer le fonctionnement du système informatique et d'autre part la nécessité d'augmenter la confiance que l'on porte sur celui-ci. Ces deux objectifs relèvent un défi important pour la recherche ; trouver des techniques et des méthodes de vérification qui permettent d'assurer le fonctionnement des systèmes d'aujourd'hui et de demain.

Théoriquement, vérifier le comportement d'un système ne pose pas de difficulté majeure. Mais en pratique, le passage d'un cas d'étude à un cas industriel pose des « problèmes d'échelle » : la taille du graphe des comportements explose de façon exponentielle. Les causes de l'explosion sont diverses et nécessitent d'être étudiées pour trouver de nouvelles méthodes permettant de les limiter.

Dans la première partie de cette conclusion nous faisons un bilan sur le travail effectué pour réduire différentes causes de l'explosion. Dans la seconde partie nous donnons un exemple de résultats obtenus avec l'outil *Tina*. Enfin dans la troisième partie nous proposons des perspectives à ce travail.

Bilan du travail effectué

L'explosion du nombre d'états peut avoir différentes causes. Nous nous sommes plus particulièrement intéressés à deux d'entre elles :

- en atemporel : l'explosion due à la représentation du parallélisme par l'entrelacement d'actions
- en temporel : l'explosion due aux contraintes temporelles

En atemporel

Cette thèse s'est focalisée sur l'utilisation des ordres partiels pour réduire l'explosion combinatoire due à la représentation du parallélisme par l'entrelacement d'actions.

L'idée générale qui a guidé notre travail peut se résumer par : « Trouver la meilleure réduction possible pour une classe de propriétés ». En effet, l'utopie de vouloir réduire sans rien perdre quant au pouvoir de vérification n'est malheureusement pas atteignable. La construction d'un graphe réduit sera d'autant plus efficace que l'on accepte de perdre de l'information. Nous avons donc travaillé et proposé plusieurs approches en considérant différentes classes de propriétés.

Un premier travail pour préserver les propriétés de blocage a été mené. Il nous a permis de définir une technique, les graphes de pas persistants, permettant de faire collaborer deux techniques (les ensembles persistants et les graphes de pas couvrants). Nous avons montré que cette méthode était une généralisation des deux méthodes la constituant. Les résultats obtenus expérimentalement montrent qu'elle permet dans bien des cas de fonctionner aussi bien que la

meilleure des deux techniques. Ce résultat est important d'un point de vue pratique, puisque au lieu d'essayer une première technique puis si elle n'aboutit pas (mais comment savoir qu'elle n'aboutit pas ?) on essaye avec une seconde, ici on essaye une seule technique et elle fait le mieux possible. De plus elle permet même parfois la construction d'un graphe plus petit que celui construit par les deux autres.

Nous nous sommes également intéressés à une classe de propriétés plus larges en considérant les propriétés invariantes par bégayement dont toutes les formules de LTL_{-X} font partie. Nous avons montré qu'on pouvait construire un graphe réduit préservant ces propriétés en utilisant des pas. Pour cela on s'autorise à construire des pas sauf sur les transitions qui peuvent changer la valuation de propositions atomiques présentes dans la formule ; pour ces transitions on utilise l'exploration classique. Cette technique permet une réduction conséquente pour certains systèmes. La taille des graphes obtenus a été comparée à ceux obtenus par la technique des ensembles amples qui permet également la préservation de formules de LTL invariantes par bégayement. Cette comparaison a montré que les deux approches étaient très complémentaires puisqu'elles permettent généralement d'obtenir une réduction significative sur des systèmes différents.

En temporel

Les contraintes temporelles sont une cause directe de l'explosion du nombre d'états lors de l'analyse de systèmes temporisés. Nous avons donc essayé d'attaquer le problème directement à sa source en essayant de définir une technique de sur-approximation du comportement permettant de relâcher certaines contraintes temporelles. La définition d'une telle sur-approximation a été possible en exploitant la propriété d'absence de blocage temporel. Cette propriété est une caractéristique fondamentale des systèmes de transitions à intervalles de temps sur les transitions et une sémantique forte obligeant le tir d'une transition avant la fin de son intervalle.

Nous avons décrit une procédure de décision semi effective pour vérifier des propriétés de LTL sur une sur-approximation du système. Cette approche permet dans de nombreux cas la construction d'un graphe plus petit permettant de vérifier la formule.

La plus grande sur-approximation qui peut être obtenue par cette approche est obtenue en relâchant toutes les contraintes maximales et minimales. Cette sur-approximation correspond à une exploration atemporelle (au détail près de la divergence temporelle). Un intérêt de cette approximation est donc de pouvoir exploiter les techniques de réduction existantes et développées dans la première partie de la thèse : en fonction de la propriété que l'on cherche à vérifier les ordres partiels permettent la construction d'un graphe réduit. La propriété peut alors être vérifiée sur le graphe réduit de la sur-approximation du système.

Lorsque la sur-approximation ne permet pas de conclure, nous avons défini une technique de « guidage » permettant l'utilisation des résultats trouvés sur la sur-approximation pour analyser le système temporel. Ce guidage permet dans de nombreux cas de limiter l'exploration du système temporel et donc de limiter l'explosion due aux contraintes temporelles. A partir de cette approche nous avons également défini une variante permettant la vérification par bandes : la vérification ne nécessite plus la construction d'un seul graphe (réduit par rapport à l'approche classique), mais la construction de plusieurs graphes potentiellement plus petits.

Implémentation

Les différentes techniques discutées et développées au cours de cette thèse ont été implémentées dans l'outil (*Tina*) développé au LAAS, qui permet l'analyse de réseaux de Petri. Les options de constructions de graphes réduits sont donc les suivantes :

– Ensembles persistants (heuristique du min) GP_{min}

- Graphe de pas persistant (avec différentes heuristiques $GP_{max}P$,GPP_{min},$GPPH$)
- Graphe de pas couvrants
- Graphe de pas couvrants préservant l'équivalence observationnelle
- Graphe de pas couvrants préservant l'équivalence de refus
- Ensembles proviso
- Ensembles amples
- Graphe de pas couvrants préservant LTL

La vérification de propriétés LTL par model checking a été implémentée dans un outil appelé *Mc*. L'intérêt d'une vérification a posteriori est de permettre la vérification de nombreuses propriétés en ayant calculé une seule fois le graphe d'accessibilité. Le fonctionnement de l'outil est décrit dans la partie I.3, tandis que l'annexe E en donne les différentes options. On note en particulier la possibilité d'obtenir un contre-exemple pouvant être chargé dans le simulateur de *Tina*. Cet outil sera ajouté à la distribution de la boîte à outils *Tina* très prochainement.

La technique de vérification par sur-approximation et de vérification par bandes a été implémentée (à partir du code de *Tina*) dans un prototype pour la vérification des états de blocages. Le prototypage de ces techniques a constitué une étape importante pour pouvoir tester, comparer et améliorer les différentes approches proposées.

Notre participation au projet *COTRE* (Composants Temps Réel) nous a aussi permis d'éprouver nos techniques et leur implémentation dans le cadre d'un projet industriel. Le projet *COTRE* était un projet RNTL exploratoire regroupant la fédération Feria (ONERA-DTIM, LAAS et IRIT), mais également EADS AIRBUS-France (maître d'oeuvre), TNI-Valiosys et l'ENST de Bretagne. L'objectif de ce projet exploratoire était de définir une démarche outillée pragmatique de modélisation et de validation d'architecture de logiciels temps réel permettant de combiner les approches formelles et semi-formelles dans un processus industriel garantissant la continuité/traçabilité de la phase de conception jusqu'à celle d'implantation du logiciel sur la cible réelle. Une plate-forme, intégrant les outils de description et de vérification, a été développée dans le cadre de ce projet. Pour cela nous avons défini un langage de description commun aux différents formalismes utilisés pour vérifier le système. L'annexe B décrit les grands lignes de ce projet.

Perspectives

Le travail présenté dans cette thèse a permis d'obtenir certains résultats, mais nous ouvre surtout beaucoup de perspectives pour prolonger ce travail.

Combinaison de techniques

Les graphes de pas couvrants étendus aux formules bégayement invariantes et les ensembles amples obtiennent des résultats de réduction différents suivant les systèmes considérés. Ce résultat n'est pas surprenant puisque chacune de ces techniques utilise des stratégies différentes et même orthogonales pour réduire la taille du graphe.

Comme nous l'avons fait dans le cadre de la préservation des états de blocage, avec les ensembles persistants et les graphes de pas couvrants, nous pensons qu'il peut être intéressant de faire collaborer ces deux techniques. Cependant la définition d'une collaboration et la preuve que cette collaboration permet également de préserver les propriétés de LTL invariantes par bégayement restent à faire.

Modularité de l'implémentation

Tina offre d'ores et déjà la possibilité d'explorer l'espace d'états suivant deux politiques globales : profondeur ou largeur d'abord. Cependant, au cours de l'implémentation des différents algorithmes, nous avons été amenés à considérer - indépendamment de ce choix de politique globale - des stratégies complémentaires d'exploration.

Une amélioration possible de *Tina* serait de réfléchir à une interface permettant de paramétrer facilement ces stratégies. La réflexion pour définir une telle interface reste à mener. Une attention toute particulière doit être donnée à l'influence sur les performances, car c'est une partie de code critique pour les performances de l'exploration. Nous donnons ci-dessous quelques arguments en faveur d'un paramétrage de l'exploration de l'espace d'états.

Pour implémenter les techniques par choix ("ensembles persistants", ...) nous avons codé en dur les stratégies permettant de sélectionner le sous-ensemble de transitions sensibilisées qui serait effectivement exploré. Il serait intéressant de définir une politique de sélection plus générique que l'on pourrait par exemple appliquer lors d'explorations à une symétrie près.

Dans la pratique, de nombreuses propriétés peuvent être vérifiées au vol : ainsi si l'on cherche à vérifier qu'un système n'est pas bloquant, on peut vouloir arrêter dès que l'on a trouvé un état de blocage (on peut aussi préférer les obtenir tous). Il serait donc utile de pouvoir stopper l'exploration dès qu'une configuration spécifique (qu'il faudrait spécifier) a été atteinte.

Lors d'une vérification au vol, on souhaite idéalement explorer prioritairement les parties du graphe permettant de conclure à la satisfaction ou la non satisfaction de la formule. Bien que ce problème soit indécidable, on peut envisager des heuristiques qui permettent d'obtenir de bons résultats dans de nombreux cas. Ainsi pour décider de la quasi-vivacité d'un réseau de Petri (cf III.7), on peut arrêter dès que l'on a sensibilisé toutes les transitions du réseau de Petri. Pour tenter d'accélérer leur découverte, une heuristique a consisté à explorer prioritairement les états issus de transitions non encore explorées. Une autre heuristique aurait été d'explorer prioritairement les transitions non encore explorées. On peut imaginer que suivant le type de propriétés considéré, on sera amené à mettre en place des heuristiques différentes (on pourrait aisément améliorer le guidage proposé en section VI.2 en favorisant les transitions conduisant plus rapidement atemporellement aux blocages). Il serait donc intéressant là aussi de pouvoir définir de façon générique ce type d'heuristique.

Pour différentes raisons, nous avons eu recours à la construction de produits synchrones : vérification de formules *LTL* en synchronisant l'automate de Büchi associé dans le chapitre I.3, ou au chapitre VI.2 lors du "guidage" permettant de réduire l'exploration temporelle. Dans les deux cas, cette synchronisation a été faite de façon ad hoc et mériterait d'être factorisée. Parmi les autres applications possibles, on peut penser à son utilisation lorsque les propriétés à vérifier ont été décrites par un observateur (tels les dépassements de "deadline" dans les traductions *COTRE*).

Généralisation : Extension aux systèmes de transitions généralisées

Le formalisme de modélisation utilisé dans *Tina* était jusqu'à présent les réseaux de Petri temporels (place-transitions) et ne permettait pas de manipuler explicitement des données. Si la modélisation d'un compteur est réalisable aisément, modéliser un type énuméré ou une pile est relativement complexe et permet de toucher rapidement les limites d'une modélisation dans un formalisme de bas niveau. Fort de l'expérience avec le projet *COTRE*, nous avons cherché à avoir une modélisation plus intuitive qui nécessite moins de traductions. Nous nous sommes donc intéressés aux systèmes de transitions prédicat/actions temporisées qui permettent de prendre en compte des données.

L'application des techniques de réduction « ordre partiel » que nous avons proposées à des formalismes de description de haut-niveau (i.e prenant en compte les données) est loin d'être immédiate sauf à « déplier » - lorsque c'est possible - la description de haut-niveau et ensuite à appliquer les techniques classiques sur la traduction obtenue. Appliquer nos techniques de réduction directement sur une formalisation de haut-niveau nécessite de redéfinir la relation d'indépendance et d'en proposer une approximation facile à calculer.

Le choix le plus simple consiste à décider de l'indépendance de deux transitions dès que celles-ci n'opèrent sur aucune variable commune. Si cette approximation sera facile à établir syntaxiquement, elle nous semble grossière et mérite d'être affinée. Comme dans le cas classique, plus l'approximation sera fine et plus l'espoir de réduction sera grand. Il nous semble donc intéressant d'essayer de trouver une approximation de la relation d'indépendance plus fine et ne pas écarter d'emblée de la relation d'indépendance deux transitions opérant sur la même variable.

```
   (t1) x > 0 -> (x := x + 1, z := 4 )
et
   (t2) x > 0 -> ( x := x + 2 , y := 3)
```

A titre d'exemple : $t1$ et $t2$ partagent la variable x et seront pour autant indépendantes.

La prise en compte des données introduit une nouvelle difficulté : jusqu'à présent la propriété de confluence nous assurait que deux séquences de transition ayant la même image commutative - si elles étaient sensibilisées à partir d'un même état - nous conduiraient au même état cible. Cette propriété de confluence est potentiellement perdue dès que les transitions manipulent des données. Il suffit de considérer deux transitions travaillant sur une même variable par le biais d'opérations non commutatives (voir par exemple les transitions $t3$ et $t4$ ci-dessous).

```
   (t3) x > 0 -> ( x := x + 1, z := 4 )
et
   (t4) x > 0 -> ( x := x * 2 , y := 3)
```

La propriété structurelle de confluence dont nous disposions pour les réseaux place-transition simplifiait la notion d'indépendance ; il suffisait en effet d'assurer que deux transitions indépendantes ne pouvaient se désensibiliser l'une-l'autre : Si $s \overset{t_1}{\rightarrow\!\!\!\!\!\rightarrow}$ et $s \overset{t_2}{\rightarrow\!\!\!\!\!\rightarrow}$ alors $s \overset{t_1.t_2}{\rightarrow\!\!\!\!\!\rightarrow}$ et $s \overset{t_2.t_1}{\rightarrow\!\!\!\!\!\rightarrow}$

Les transitions $t3$ et $t4$ vues plus haut ne se désensibilisent jamais mutuellement. Ainsi tout état sensibilisant $t3$ et $t4$ permettra le tir $t3.t4$ ou $t4.t3$, pour autant le résultat de $t3.t4$ diffère de celui de l'exécution $t4.t3$.

La monotonie est une autre propriété intéressante des réseaux place-transition qui disparaît. Les règles de sensibilisation et de tir de ces réseaux assurent que toute transition tirable pour un marquage M le sera pour un marquage $M' \geq M$. Cette propriété était elle-aussi mise à profit pour définir l'approximation de la relation d'indépendance : il suffisait d'examiner les pré-conditions des transitions sans se soucier de leur post-conditions. De plus la relation sous-jacente de « conflit potentiel » était symétrique ce qui ne sera plus le cas lorsque l'on considère les données.

```
   (t5) x < 5 -> ( y := y + 1, z := 4 )
et
   (t6) x < 8 -> ( x := x + 2 , t := 3)
```

$t6$ peut potentiellement désensibiliser $t5$ - la post-condition de $t6$ est croissante en x alors que la pré-condition de $t5$ est décroissante en x. A contrario, $t5$ ne peut inhiber $t6$: la post-condition

de $t5$ laisse invariant x donc également la pré-condition de $t6$. On voit aussi sur cet exemple que le fait qu'une transition puisse en inhiber une autre n'est pas lié au fait que ces transitions « commutent ».

Ces quelques remarques laissent penser qu'une reformulation intéressante de la relation d'indépendance pourrait s'énoncer ainsi :
Deux transitions t_1 et t_2 sont indépendantes, si et seulement si

- t_1 ne désensibilise pas t_2
- t_2 ne désensibilise pas t_1
- t_1 et t_2 confluent

Il reste à s'assurer que cette définition est pertinente et surtout à trouver des conditions permettant de décider de l'indépendance de deux transitions simplement et statiquement.

Appendix A

L'outil *Tina*

Description générale

Tina (TImed Petri Net Analyser) est un environnement logiciel permettant l'édition et l'analyse de réseaux de Petri et Temporels [MF76]. Il est disponible sur internet à l'adresse :

http ://www.laas.fr/tina

Tina n'est pas un « model-checker » au sens où il ne permet pas à lui seul (sauf bien sûr dans le cas des propriétés générales d'accessibilité) de décider de la satisfaction d'une certaine propriété. *Tina* intervient en amont du model-checker en lui fournissant un graphe d'états réduit sur lequel il pourra effectuer plus efficacement la vérification. La réduction opérée par *Tina* permet de préserver les propriétés « linéaires » ou « arborescentes » de l'espace d'états qui sera ultérieurement analysé. Afin d'obtenir une chaîne complète de « model-checking », *Tina* est couplé avec MEC [Arn87] pour la vérification de formules du μ-calcul, Aldebaran [FM91] pour la vérification de pré-ordres ou d'équivalences de comportement et nous avons écrit un module *Mc* permettant le model-checking de propriétés LTL.

Interface utilisateur

La boîte à outils *Tina* est conçue de façon modulaire. Les modules incluent :

- Un éditeur graphique (de réseaux, d'automates) incluant des fonctions de dessin de réseaux ou d'automates ;
- Des outils de construction de comportement et d'analyse structurelle ;
- Des outils de simulation de réseaux de Petri et réseaux de Petri temporels.

Ces modules peuvent être utilisés de façon combinée ou indépendante :

Utilisé seul, l'éditeur produit des fichiers utilisables ultérieurement par les outils de construction de comportement ou d'analyse. Mais les outils de construction peuvent aussi être invoqués sans sortir de l'éditeur et celui-ci permet d'éditer ou de dessiner les résultats. Les outils de simulation peuvent être invoqués directement à partir de l'éditeur, permettant de visualiser l'exécution de son modèle. L'outil de simulation est très utile pédagogiquement.

Utilisés seuls, les outils de construction et d'analyse fonctionnent comme des filtres, ils sont donc facilement insérables dans des chaînes de développement existantes. Leur ligne de commande permet de sélectionner les paramètres de construction. Ils admettent en entrée des réseaux de Petri ou Temporels décrits dans un format graphique (produit par l'éditeur), ou textuel (écrit à la main ou par programme) et produisent des résultats selon un choix de formats de sortie.

Le format textuel d'entrée est simple, intuitif, et compact. Plusieurs formats de sortie sont

disponibles. L'outil *Tina* n'impose l'usage d'aucun vérificateur de modèle spécifique. A ce jour, *Tina* produit des résultats dans divers formats, incluant un format « en clair » pour des utilisations pédagogiques, le format d'entrée « automate » de l'outil *Aldébaran* pour des analyses d'équivalences, ainsi que le format d'entrée « système de transitions » de l'outil *MEC*, pour la vérification de formules du μ-calcul. Ainsi, *Tina* peut être utilisé comme « front-end » par bon nombre d'outils de vérification, éventuellement au prix de l'écriture d'un filtre spécifique traduisant l'une des sorties possibles de *Tina* dans un format compris par l'outil de vérification utilisé.

Une capture d'écran d'une session *Tina* typique est reproduite en Figure A.1, avec un réseau temporel en cours d'édition, un résultat textuel de construction de comportement, et une représentation graphique du comportement produit.

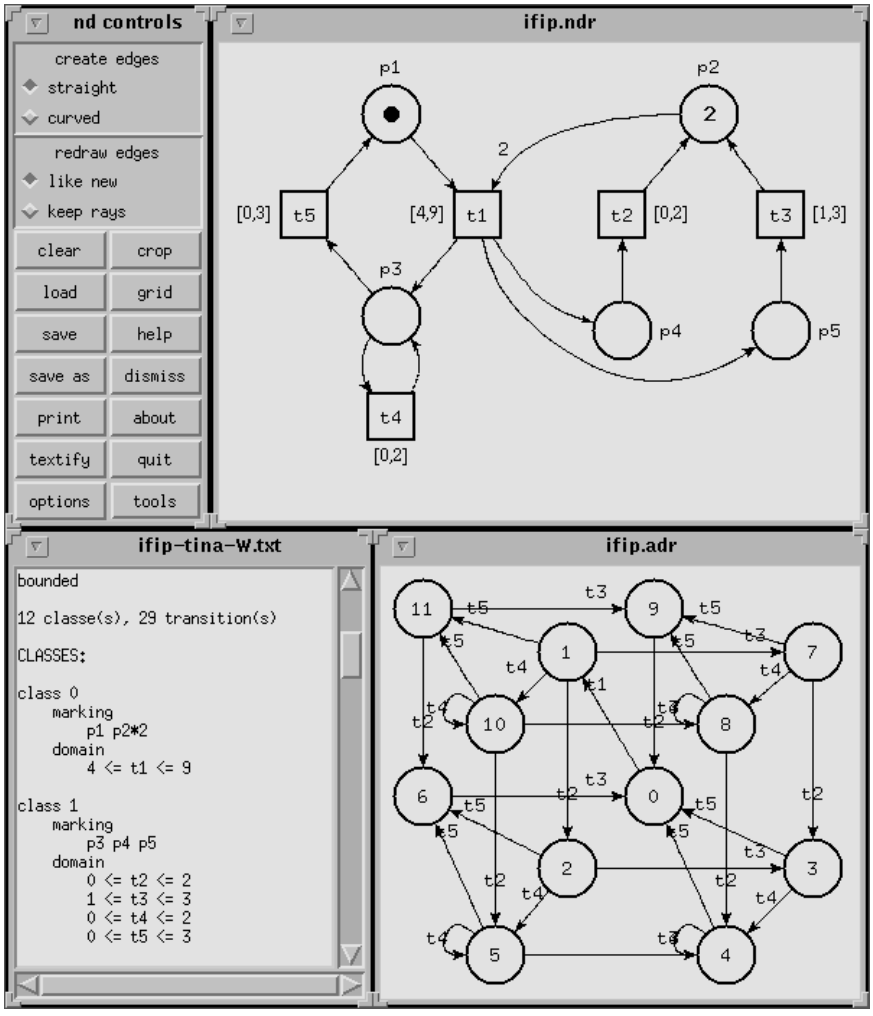


Figure A.1: Capture d'écran d'une session *Tina*

Le domaine d'utilisation de *Tina* est large. La variété de constructions qu'il propose en fait un outil utile pour l'enseignement de ces techniques. D'autre part, *Tina* est utilisé dans plusieurs projets à caractère industriel ; il figure notamment parmi les environnements de vérification retenus dans le cadre du projet RNTL COTRE (Composants Temps Réel, `http://www.laas.fr/COTRE`).

Différentes abstractions sont proposées pour permettre de vérifier différentes classes de propriétés allant des propriétés générales d'accessibilité, aux propriétés spécifiques - linéaires ou

arborescentes - exprimées par des logiques temporelles ou des équivalences de comportement. Deux grands types d'abstraction fondées sur les « ordres partiels » et les « graphes de classes » permettent respectivement de prendre en compte des systèmes atemporels ou temporisés. Dans le cas de systèmes temporisés, pouvoir considérer un espace d'états abstrait est un impératif car l'espace d'états concret est en général infini. Dans le cas de systèmes atemporels, offrir un espace d'états abstrait permet de limiter les risques d'explosion combinatoire. Différentes techniques sont offertes dans l'outil *Tina* : ensembles persistants (GP_{min}), graphes de pas couvrants, graphes de pas couvrants préservant LTL, graphes de pas persistants ($GP_{max}P$,GPP_{min},$GPPH$), graphes de pas couvrants préservant l'équivalence observationnelle, graphes de pas couvrants préservant l'équivalence de refus, ensembles proviso, ensembles amples.

Appendix B

Le projet *COTRE*

L'objectif de ce projet exploratoire était de définir une démarche outillée pragmatique de modélisation et de validation d'architecture de logiciels temps réel permettant de combiner les approches formelles et semi-formelles dans un processus industriel garantissant la continuité/traçabilité de la phase de conception jusqu'à celle d'implantation du logiciel sur la cible réelle.

Présentation

COTRE est un projet RNTL qui s'est déroulé sur deux ans, de janvier 2002 à janvier 2004. Le projet regroupait la fédération Feria (ONERA-DTIM, LAAS et IRIT), mais également EADS AIRBUS-France (maître d'oeuvre), TNI-Valiosys et l'ENST de Bretagne. Le développement de logiciels dans les systèmes avioniques embarqués est généralement divisé en quatre phases principales : analyse des besoins, conception, implémentation et test sur la machine cible. La première phase d'analyse des besoins fournit une description détaillée d'exigences exprimées de manière informelle ou dans un langage formel. L'objectif de la phase de conception et de modéliser les composants matériels et logiciels du système, du point de vue de leur comportement dynamique et de leur configuration, à en vérifier les propriétés et à déployer les fonctions logicielles sur des composants matériels.

La phase de conception part du résultat d'une analyse des besoins, c'est à dire de la description détaillée d'exigences exprimées de manière informelle ou dans un langage formel (tel que Lustre, SCADE voire SDL). Elle vise à modéliser les composants matériels et logiciels du système, du point de vue de leur comportement dynamique et de leur configuration, à en vérifier les propriétés et à déployer les fonctions logicielles sur des composants matériels.

Nous avons proposé un langage *COTRE* pour la phase de conception. Ce langage permet de décrire l'architecture des systèmes, le comportement et l'interaction de leurs composants, ainsi que les exigences fonctionnelles et non fonctionnelles sur ces systèmes. Des méthodes et des outils sont associés à ce langage et sont regroupés dans la plate-forme *COTRE* dans le but de faciliter la vérification des propriétés attendues sur le système développé et de dériver une implémentation correcte et robuste de ce système. La complexité des systèmes à représenter et la diversité des propriétés à vérifier ont amené à proposer une plate-forme ouverte permettant d'intégrer des outils associés à divers formalismes adaptés à l'expression des différents aspects à considérer : architecture, comportement et propriétés.

Caractérisation du langage *COTRE*

Le langage *COTRE* permet de décrire de manière graphique et textuelle, l'architecture de systèmes sous forme de composants matériels et logiciels, mais également d'exprimer des propriétés à vérifier. Chaque composant et l'architecture globale doivent avoir à la fois une représentation statique et dynamique. La représentation dynamique décrit un comportement fonctionnel, mais peut aussi exprimer des aspects temporels, de sûreté et de performance. Une bibliothèque décrivant les composants spécifiques de systèmes d'exploitation, en particulier ceux de la norme ARINC 653 [ARI97] d'un usage courant dans le domaine de l'avionique, fournit au concepteur un ensemble de composants réutilisables.

Les besoins exprimés ci-dessus nous ont amené à positionner le langage *COTRE* dans la famille des langages de description d'architecture (ADL) et à prendre en considération les efforts de normalisation en cours qui doivent aboutir à la définition du langage de description d'architecture pour l'avionique, AADL, proposé par la "Society of Automotive Engineers" [SAE02].

La plupart des ADLs fournissent trois concepts de modélisation de base : les composants qui correspondent aux unités de calcul ou de stockage de données, les connecteurs qui représentent les interactions entre composants et enfin les configurations, qui décrivent la structure des architectures reliant composants et connecteurs [MT00]. L'idée fondamentale qui sous-tend l'utilisation d'un ADL est que l'analyse des besoins doit produire une décomposition structurelle des systèmes en composants indépendants dont le développement peut être mené en parallèle. En conséquence, le langage *COTRE* doit supporter la description des aspects architecturaux d'un système. En outre, la décomposition modulaire doit prendre en compte les aspects réactifs et temps réel dans le comportement des composants.

Un composant *COTRE* est décrit par une spécification et une implémentation. L'implémentation dans un langage standard (C, ADA...) doit satisfaire la spécification, dans le cadre des hypothèses définies sur l'environnement. Grâce à la méthodologie ADL basée sur des composants et des connecteurs, le code écrit par les utilisateurs est essentiellement séquentiel. La partie réactive doit être vérifiée et ensuite automatiquement générée par la plate-forme de développement de logiciel *COTRE*.

Le langage *COTRE* permet également, comme le langage ACME [GMW97], de définir de nouveaux types de composants, de ports et de connecteurs, par extension de types déjà existants. De plus, contrairement à la plupart des ADLs, des modèles formels seront utilisés pour représenter des comportements. Le langage *COTRE* permet de décrire plusieurs types de propriétés non fonctionnelles et supporte différentes techniques d'analyse et de vérification de ces propriétés.

La plate-forme *COTRE*

La plate-forme logicielle *COTRE* supporte les étapes de modélisation et de vérification de la phase de conception et permet de vérifier des propriétés portant sur :

- l'utilisation de ressources comme par exemple la charge de travail d'un processeur, les débordements de mémoire-tampons ;
- l'ordonnancement de processus et de messages, comme par exemple des contraintes d'ordre d'exécution, des contraintes de dépendance, une analyse temporelle de l'ordonnancement, une analyse de sensibilité sur la durée d'un calcul ;
- des contraintes temporelles telles que des délais entre événements, des durées de traitement, des temps de réponse limites, des latences ;
- la fiabilité et la sûreté telles qu'une probabilité de faute avec recouvrement, un partitionnement temporel et spatial pour différents niveaux de sécurité ;

– des contraintes fonctionnelles telles que la correction et la robustesse.

La figure B.1 présente la plate-forme logicielle *COTRE*. Le langage *COTRE* permet, d'une part de représenter, dans une spécification C1, l'architecture du système et le comportement de ses composants et d'autre part d'exprimer, dans une spécification associée C2, les propriétés attendues des composants et leur environnement présumé.

Des spécifications formelles FC1 et FC2, utilisées pour la vérification sont extraites des spécifications C1 et C2, en prenant en compte les propriétés à vérifier (propriétés temporelles, de sûreté...). Les spécifications peuvent être améliorées à partir des résultats de simulation, de vérification ou d'évaluation, soit au niveau du langage *COTRE* dans les représentations C1 et C2, soit dans les représentations formelles FC1 et FC2. Après vérification d'une spécification, des outils de génération de jeux de tests pourront être utilisés pour préparer les tests d'une future implémentation correspondant à cette spécification.

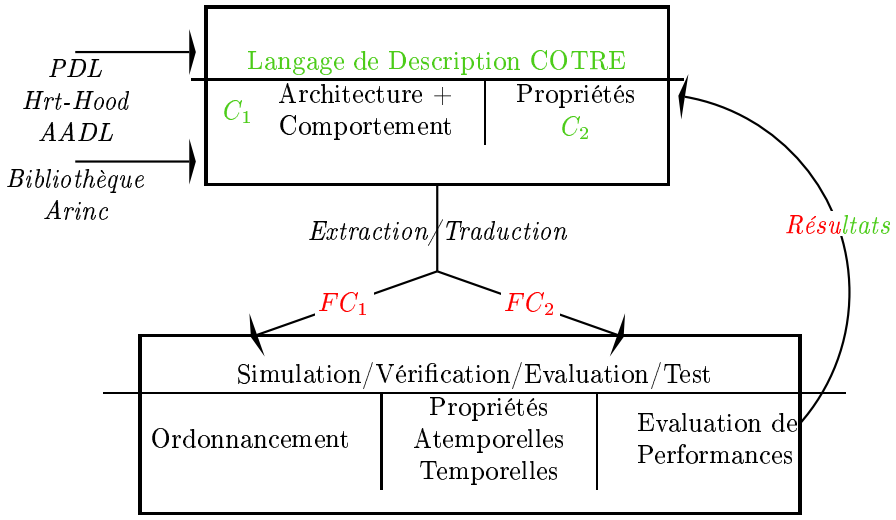


Figure B.1: Architecture générale de la plate-forme logicielle *COTRE*

Les techniques de vérification peuvent être basées sur la satisfaction de formules ou sur la comparaison de modèles :

– Quand les propriétés souhaitées sont représentées par un comportement abstrait (c'est le cas des propriétés de l'environnement) la vérification est basée sur l'équivalence (ou l'inclusion) entre le système concret et l'une de ses abstractions. Différents pré-ordres ou équivalences peuvent être considérés selon la classe de propriétés (équivalence de langage, sémantique du refus, bisimulation...). Des outils disponibles comme Aldebaran [FM91] permettent de telles vérifications. Pour les systèmes temporisés, des outils comme Tina [BV03] ou Minim [TY96] permettent d'obtenir des bisimulations en faisant abstraction du temps (« Time-abstracting bisimulations »).

– Quand les propriétés souhaitées sont exprimées au moyen de formules de logique temporelle, la vérification du système consiste à vérifier que le système est un modèle d'un ensemble spécifique de formules. Différentes logiques temporelles sont envisageables : qualitative comme LTL [Pnu86] ou CTL [CES86] , ou quantitative comme leurs extensions temporisés telles que TCTL [ACD90]. Les environnements tels que SMV [BCM+90], Uppaal [PL00] et Kronos [Yov97] ont été considérés.

Dans beaucoup de cas, l'explosion de l'espace d'états est un problème important. L'intégration des techniques de réduction basées sur les ordres partiels dans l'outil *Tina* a permis de limiter

cette explosion. Cependant seules les techniques de réductions atemporelles étaient développées à l'époque du projet, ce qui nécessitait des traductions pour représenter le comportement temporel, par un sur-comportement atemporel fini. En effet la suppression simple des contraintes temporelles rendaient le plus souvent le modèle non borné, par exemple à cause du motif de « générateur de période du processus ».

Déroulement du projet

Les travaux et résultats de ce projet sont présentés dans différents rapports de contrats. Nous avons d'abord commencé par un état des lieux des différentes techniques et outils disponibles qui pouvaient être utiles au projet. Le rapport [ABB+02] fait un résumé des concepts et outils, tandis que le rapport [BBD+03c] concerne plus particulièrement les outils de vérification. Nous avons alors travaillé sur la définition de la plate-forme complète pour voir comment les différents outils pouvaient collaborer le plus simplement possible pour l'utilisateur. [BBD+03a] montre comment ces outils peuvent être utilisées dans le cadre du projet. L'étape suivante a consisté à valider des cas d'études fournis par le maître d'oeuvre ([BBD+03b]).

Une part importante du projet a alors consisté dans la définition d'un langage de description commun appelé **V-COTRE**. L'objectif de ce langage est d'avoir une description unique de référence du système qui soit ensuite traduit dans différents formalismes de vérification (réseaux de Petri, automates...). La difficulté était donc d'avoir un langage suffisamment général qui puisse englober toutes les informations utiles pour les vérifications ultérieures. La sémantique du langage devait être complète pour ne pas laisser la place à des interprétations ultérieures.

L'étape suivant était la traduction d'une spécification **V-COTRE** dans un formalisme de vérification. En pratique cette étape a été entrelacée avec la précédente, car nous avons fait de nombreux aller-retour lorsque les définitions étaient incomplètes, ou imprécises. J'ai plus particulièrement travaillé sur la traduction de **V-COTRE** vers les réseaux de Petri et réseaux de Petri temporels[BBD+04].

Enfin les prospectives du projets sont présentées dans [BBF+04].

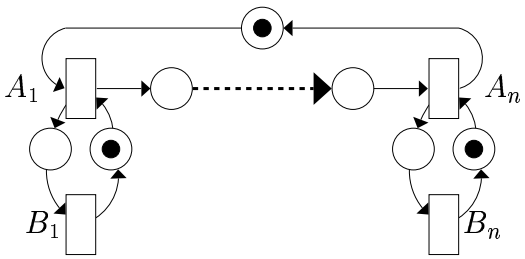
Appendix C

Exemples traités dans la thèse

Différents exemples ont été utilisés dans les chapitres précédents. Un même système peut être modélisé de différentes façon, nous précisons donc ici la modélisation qui a été utilisée.

Le scheduler de Milner

Scheduler de Milner [Mil85] : n sites exécutent cycliquement l'action A_i puis l'action B_i. Un scheduler contraint l'exécution du système complet de telle sorte que les n sites alternativement exécutent les actions $A_1, A_2, \ldots A_n$. Le réseau ci-contre représente le système complet (scheduler + n sites).



La Figure C.1 représente le STE associé dans le cas de 2 sites.

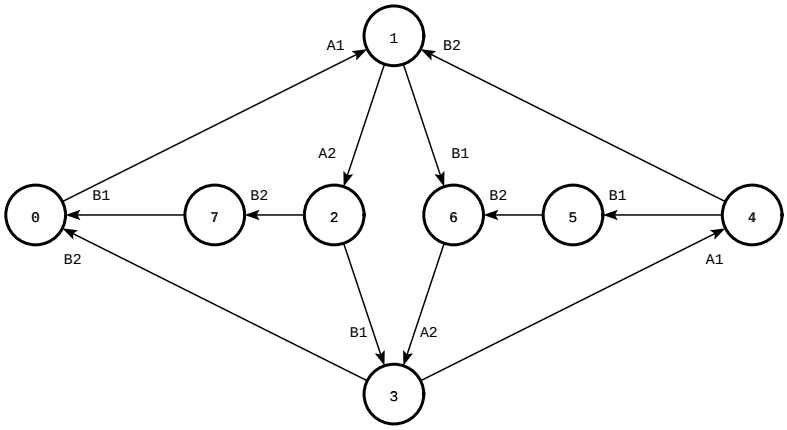


Figure C.1: Scheduler de Milner - Graphe d'accessibilité pour $n = 2$

La base de données

Le modèle de la base de données a été présenté dans [Jen86] pour illustrer les méthodes de réduction d'espace d'états basées sur les symétries. Ce même exemple a été repris dans [Val89] pour illustrer les réduction d'espace d'états basées sur les ensembles stubborns.

La base est constituée par $n \geq 2$ gestionnaires et un mécanisme d'exclusion mutuelle. Initialement, tous les gestionnaires sont oisifs et la base est ouverte. L'écriture d'une donnée par le gestionnaire k est modélisée par une transition « update and send messages » ($usm(k)$) où une requête est envoyée à tous les autres gestionnaires. Le gestionnaire actif (k) attend alors un acquittement de chacun des autres gestionnaires. Le gestionnaire p recevant la requête $usm(k)$(transition $rm(k,p)$) effectue la mise à jour puis retourne un acquittement (transition $sa(k,p)$). Lorsque tous les acquittements sont disponibles le gestionnaire actif redevient oisif et la base est de nouveau ouverte ($ra(k)$).

La figure C.2 représente la forme générale de l'espace d'états dans le cas de n gestionnaires. Chaque $Cube_k$ représente l'exécution entrelacée des actions $rm(k,p)$ et $sa(k,p)$ pour $p \in [1,N], p \neq k$. La figure C.3 représente le « cube » 1 dans le cas de 3 gestionnaires. Pour ce système, les couples de transition en conflit ont la forme suivante : (t,t) ou $(usm(i), usm(j))$ ou $(usm(i), rm(i,j))$.

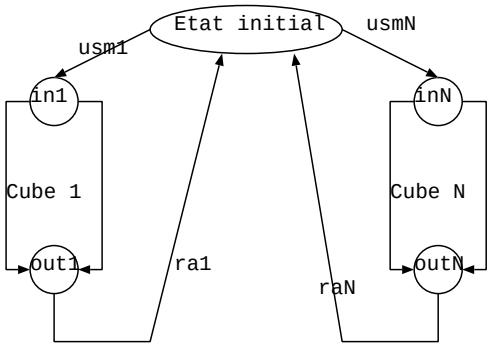


Figure C.2: Base de données - Forme générale du STE

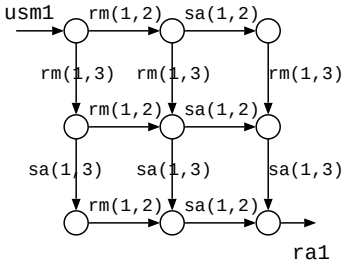


Figure C.3: Base de données - Cube 1 pour 3 gestionnaires

La piscine

L'exemple de la piscine a été présenté dans [BF99].

La transition $T1$ représente l'arrivée d'un client qui prend une cabine libre (place $x6$) et se déshabillent en place $x1$. Puis il prend un panier (transition $T2$). Les paniers libres sont comptés dans la place $x7$. Après avoir mis ses affaires dans le panier, le client libère la cabine (un jeton est remis en place $x6$) et va dans le bassin. À la sortie du bassin, transition $T4$ le client reprend une cabine libre. Après s'être changé il libère le panier, qui revient dans le stock des paniers libres (place $x7$). Puis il quitte la piscine et libère la cabine.

x1 : déshabille x2 : rangement des habits dans le panier x3 : bain x4 : prendre les habits du panier x5 : s'habiller x6 : cabines libres x7 : paniers libres r1 : arrivée des clients r6 : départs des clients

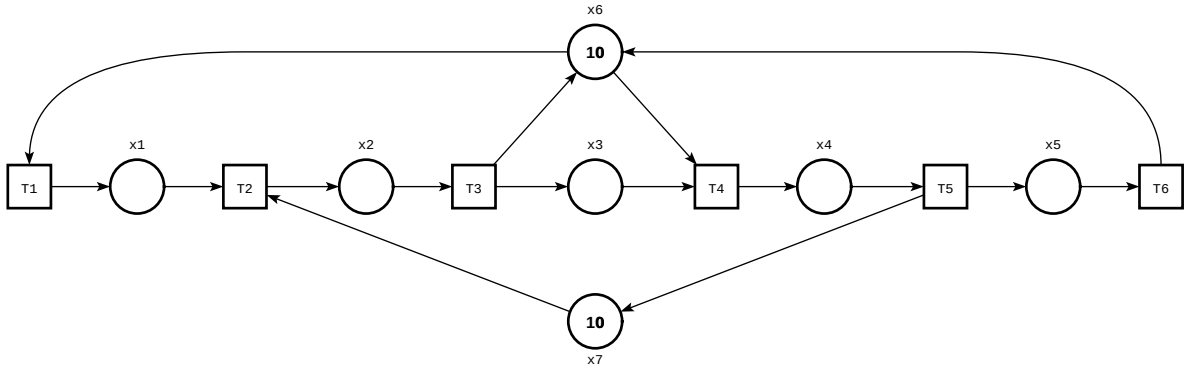


Figure C.4: La piscine

Les vérifications de ce modèle montrent que quelque soit la valeur k du nombre de cabines et de paniers, il peut y avoir un interblocage : Une arrivée massive de clients peut occuper toutes les cabines, mais ces clients seront en attente d'un panier libre. Si plus aucun panier n'est libre, il faut attendre que certains d'entre eux se libèrent, mais dans le cas où tous les clients présents actuellement étaient en train de se baigner, alors aucun ne peut quitter la piscine puisque aucune cabine n'est libre.

Naimi Trehel

L'algorithme développé est une variante de l'algorithme de Naimi-Trehel [NT87, NTA96], un algorithme distribué d'exclusion mutuelle. Suivant la taxonomie des algorithmes d'exclusion mutuelle proposée dans [Ray91], l'algorithme étudié appartient à la famille des algorithmes à jeton et plus particulièrement à ceux utilisant une arborescence distribuée dynamique. La position de la racine dans l'arborescence symbolise la possession du jeton. Dans les algorithmes à base de jeton, c'est la possession du jeton qui donne l'autorisation d'accès à la section critique. Pour les algorithmes utilisant une arborescence, un processus envoie sa requête à un voisin qualifié, son « père ». Dans l'algorithme de Naimi-Trehel et la variante que nous utilisons, cette requête « remonte » l'arborescence des processus jusqu'à celui détenant le jeton qui l'envoie au demandeur.

Une particularité de l'algorithme de Naimi-Trehel est que l'arborescence des sites est dynamique. Chaque réception de requête modifie le père du récepteur, le nouveau père du récepteur étant l'émetteur de la requête traitée.

Buffers asynchrones

Ce modèle correspond à un algorithme de communication par buffers asynchrones. Le processus de gauche est un générateur d'éléments, tandis que les autres processus reçoivent la valeur du côté gauche, font un calcul interne et (mis à part le processus de droite) envoie la nouvelle valeur au processus de droite.

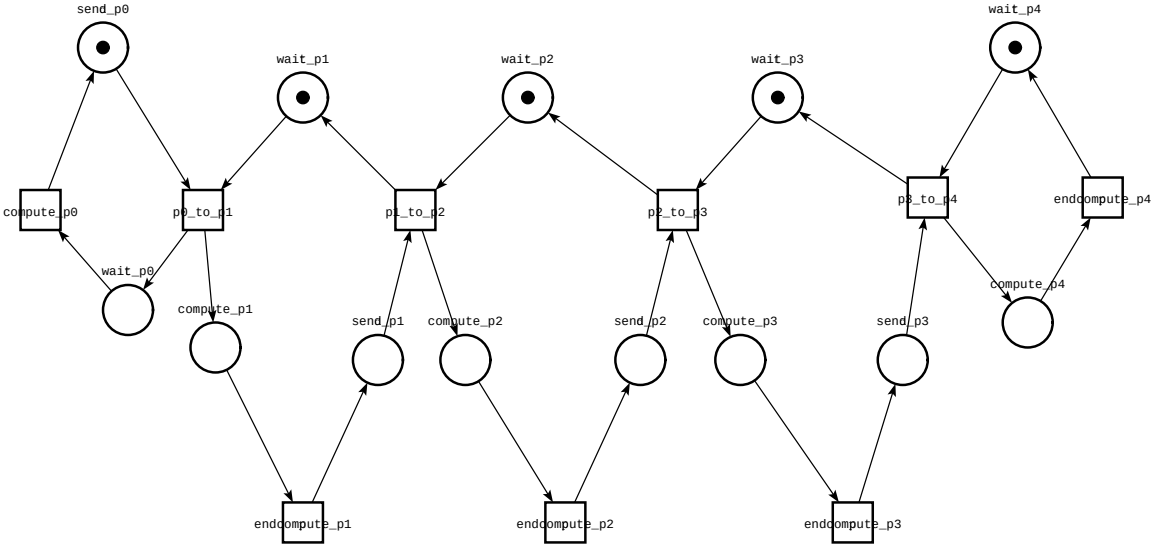


Figure C.5: Exemple des buffers asynchrones

Les philosophes

Les philosophes sont devenus un classique. Chaque philosophe a trois états dans son comportement : « penser », « manger » et « attendre fourchette ». Ce modèle peut arriver à un interblocage ; si chaque philosophe souhaite manger et a déjà pris sa fourchette de droite, tous les philosophes sont en attente d'une fourchette qui ne sera jamais libérée. D'autre part si on n'ajoute pas une notion d'équité, il peut se produire un phénomène de « famine » : un philosophe attend indéfiniment ses fourchettes pendant que d'autres philosophes pensent puis mangent indéfiniment ; le système n'est pas bloqué, mais le philosophe ne passe jamais à l'état mange.

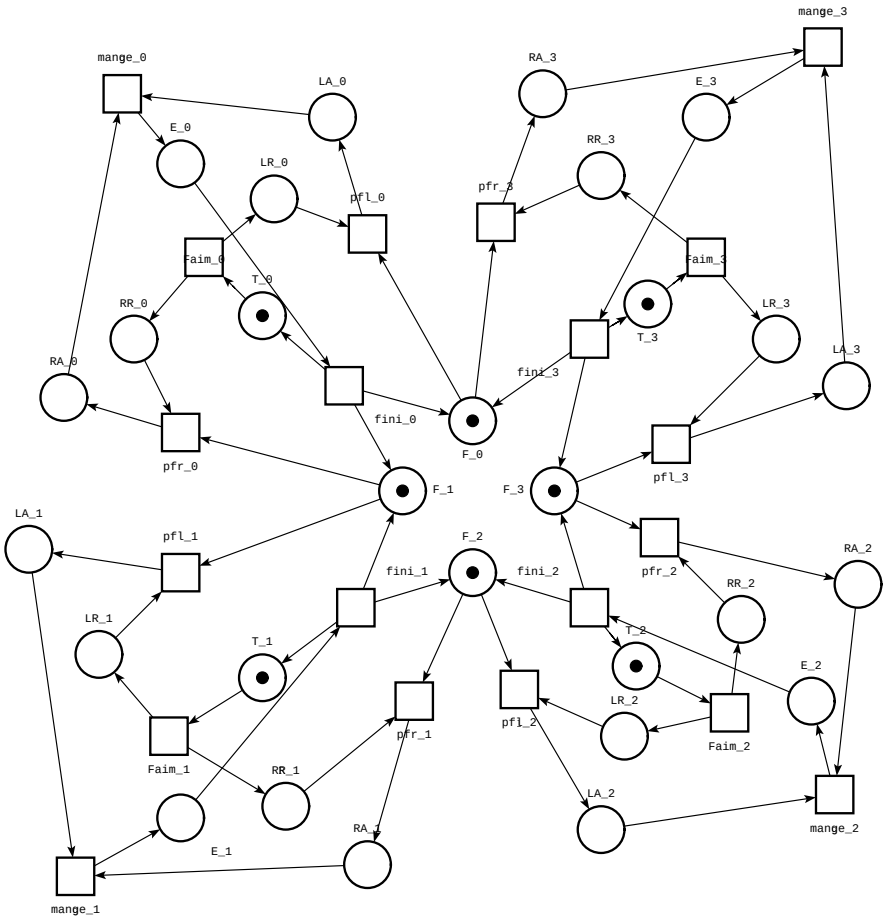


Figure C.6: Modèle des Philosophes

La figure C.6 donne une modélisation du problème des philosophes en réseau de Petri pour 4 philosophes.

Le cueilleur de bananes

Cet exemple est une variante de « la journée du planteur de bananes » de [Had01]. Le réseau modélise (de façon peu réaliste) le travail d'un cueilleur de bananes. Initialement, sur son champ (supposé de capacité infinie), le planteur cueille des bananes, puis rentre avec ses bananes et se met à table pour en manger quelques-unes. Une fois son repas terminé, il se lève et jette quelques peaux de banane dans son jardin avant d'aller dormir. Il revient alors au champ pour travailler.

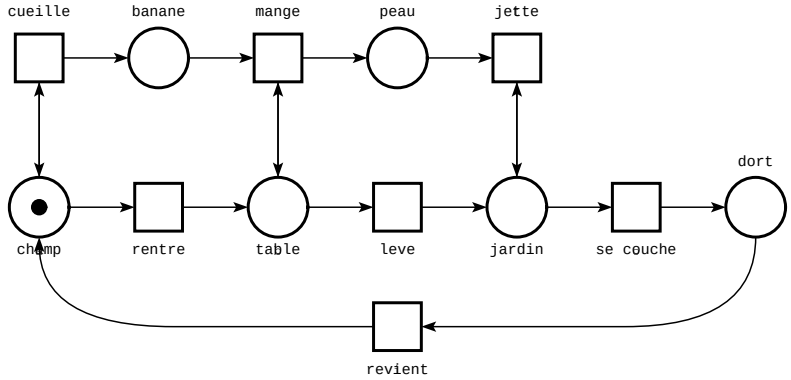


Figure C.7: Le cueilleur de bananes

Token Ring : protocole d'exclusion mutuelle par jeton tournant

Le token ring est un algorithme distribué standard d'exclusion mutuelle pour lequel n utilisateurs organisés en anneau se passent un jeton symbolisant le droit d'accéder à une ressource critique. La figure C.8 donne le réseau de Petri représentant ce système pour 4 utilisateurs. Chaque utilisateur peut soit prendre le jeton, soit passer le jeton directement à l'utilisateur suivant.

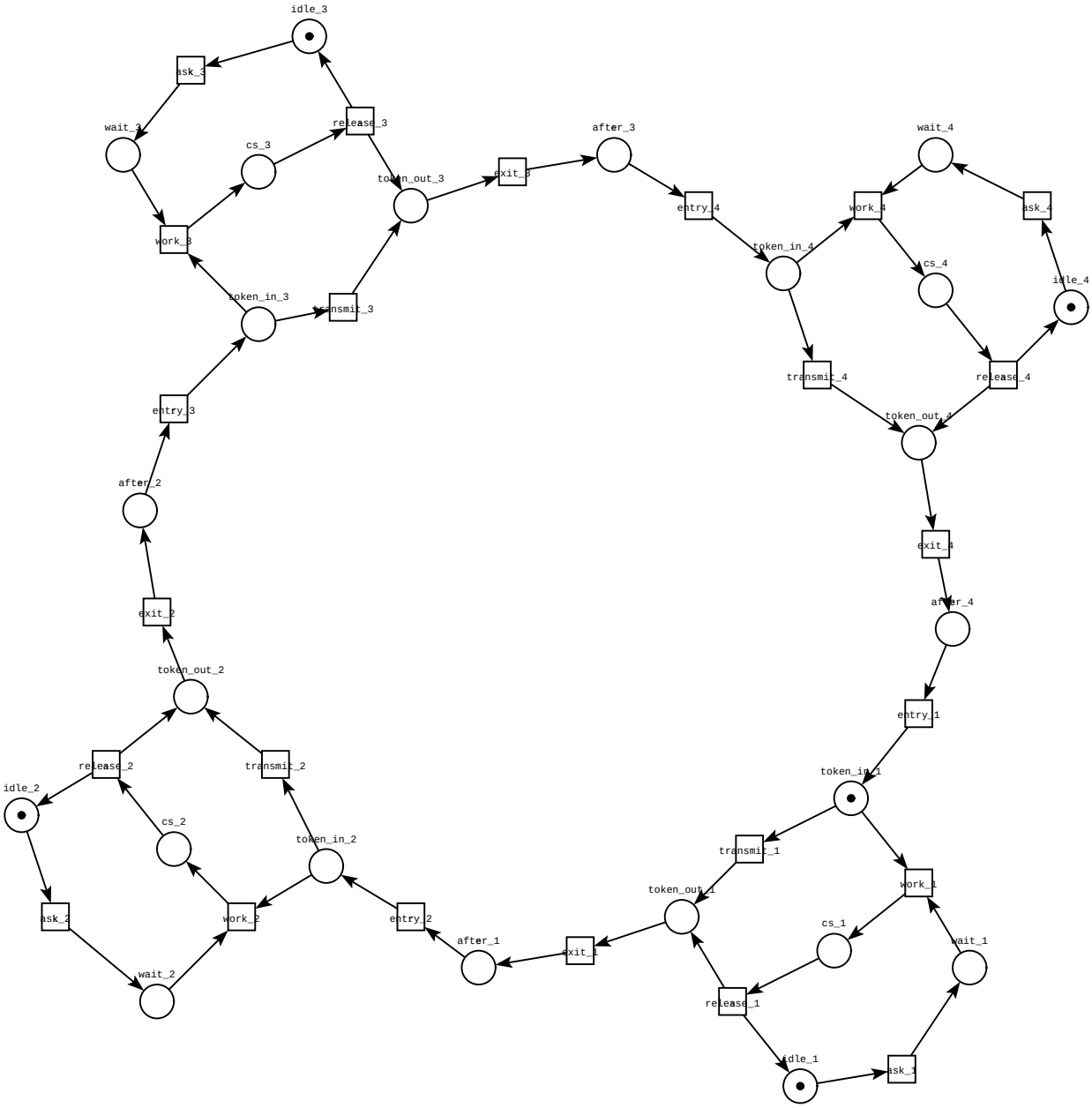


Figure C.8: Token ring avec 4 utilisateurs

Appendix D

Contre-exemple donné par l'outil *Mc*

Nous donnons dans cette annexe, le contre-exemple complet donné par une vérification d'un token ring à 4 stations avec l'outil *Mc*. Le contre-exemple condensé a été donné dans la partie 3.4.2 du chapitre I.

```
>mc tokenring04.ktz
Mc version 2.7.4 -- 03/29/05 -- LAAS/CNRS
ktz loaded, 224 states, 688 transitions
0.020s

-  infix x suivipar y = [] ( x => () <> y);
infix operator suivipar : prop \# prop -> prop
0.000s


- output fullproof;
output mode set
0.000s

-  wait_1 suivipar work_1;
FALSE
  state 0: idle_1 idle_2 idle_3 idle_4 token_in_1
  -transmit_1->
  state 1: idle_1 idle_2 idle_3 idle_4 token_out_1
  -exit_1->
  state 2: after_1 idle_1 idle_2 idle_3 idle_4
  -entry_2->
  state 3: idle_1 idle_2 idle_3 idle_4 token_in_2
  -ask_1->
  state 4: idle_2 idle_3 idle_4 token_in_2 wait_1
  -transmit_2->
  state 5: idle_2 idle_3 idle_4 token_out_2 wait_1
  -exit_2->
  state 6: after_2 idle_2 idle_3 idle_4 wait_1
  -entry_3->
```

```
state 7: idle_2 idle_3 idle_4 token_in_3 wait_1
-transmit_3->
state 8: idle_2 idle_3 idle_4 token_out_3 wait_1
-exit_3->
state 9: after_3 idle_2 idle_3 idle_4 wait_1
-entry_4->
state 10: idle_2 idle_3 idle_4 token_in_4 wait_1
-transmit_4->
state 11: idle_2 idle_3 idle_4 token_out_4 wait_1
-exit_4->
state 12: after_4 idle_2 idle_3 idle_4 wait_1
-ask_4->
state 13: after_4 idle_2 idle_3 wait_1 wait_4
-entry_1->
state 14: idle_2 idle_3 token_in_1 wait_1 wait_4
-work_1->
state 15: cs_1 idle_2 idle_3 wait_4
-release_1->
state 16: idle_1 idle_2 idle_3 token_out_1 wait_4
-exit_1->
state 17: after_1 idle_1 idle_2 idle_3 wait_4
-entry_2->
state 18: idle_1 idle_2 idle_3 token_in_2 wait_4
-transmit_2->
state 19: idle_1 idle_2 idle_3 token_out_2 wait_4
-exit_2->
state 20: after_2 idle_1 idle_2 idle_3 wait_4
-entry_3->
state 21: idle_1 idle_2 idle_3 token_in_3 wait_4
-transmit_3->
state 22: idle_1 idle_2 idle_3 token_out_3 wait_4
-exit_3->
state 23: after_3 idle_1 idle_2 idle_3 wait_4
-entry_4->
state 24: idle_1 idle_2 idle_3 token_in_4 wait_4
-work_4->
state 25: cs_4 idle_1 idle_2 idle_3
-release_4->
state 26: idle_1 idle_2 idle_3 idle_4 token_out_4
-exit_4->
state 27: after_4 idle_1 idle_2 idle_3 idle_4
-ask_4->
state 28: after_4 idle_1 idle_2 idle_3 wait_4
-entry_1->
state 29: idle_1 idle_2 idle_3 token_in_1 wait_4
-ask_2->
state 30: idle_1 idle_3 token_in_1 wait_2 wait_4
-transmit_1->
```

```
state 31: idle_1 idle_3 token_out_1 wait_2 wait_4
-exit_1->
state 32: after_1 idle_1 idle_3 wait_2 wait_4
-entry_2->
state 33: idle_1 idle_3 token_in_2 wait_2 wait_4
-work_2->
state 34: cs_2 idle_1 idle_3 wait_4
-ask_3->
state 35: cs_2 idle_1 wait_3 wait_4
-release_2->
state 36: idle_1 idle_2 token_out_2 wait_3 wait_4
-exit_2->
state 37: after_2 idle_1 idle_2 wait_3 wait_4
-entry_3->
state 38: idle_1 idle_2 token_in_3 wait_3 wait_4
-work_3->
state 39: cs_3 idle_1 idle_2 wait_4
-ask_2->
state 40: cs_3 idle_1 wait_2 wait_4
-release_3->
state 41: idle_1 idle_3 token_out_3 wait_2 wait_4
-exit_3->
state 42: after_3 idle_1 idle_3 wait_2 wait_4
-entry_4->
state 43: idle_1 idle_3 token_in_4 wait_2 wait_4
-work_4->
state 44: cs_4 idle_1 idle_3 wait_2
-release_4->
state 45: idle_1 idle_3 idle_4 token_out_4 wait_2
-exit_4->
state 46: after_4 idle_1 idle_3 idle_4 wait_2
-entry_1->
state 47: idle_1 idle_3 idle_4 token_in_1 wait_2
-transmit_1->
state 48: idle_1 idle_3 idle_4 token_out_1 wait_2
-exit_1->
state 49: after_1 idle_1 idle_3 idle_4 wait_2
-entry_2->
state 50: idle_1 idle_3 idle_4 token_in_2 wait_2
-work_2->
state 51: cs_2 idle_1 idle_3 idle_4
-release_2->
state 52: idle_1 idle_2 idle_3 idle_4 token_out_2
-exit_2->
state 53: after_2 idle_1 idle_2 idle_3 idle_4
-entry_3->
state 54: idle_1 idle_2 idle_3 idle_4 token_in_3
-ask_2->
```

```
state 55: idle_1 idle_3 idle_4 token_in_3 wait_2
-transmit_3->
state 56: idle_1 idle_3 idle_4 token_out_3 wait_2
-exit_3->
state 57: after_3 idle_1 idle_3 idle_4 wait_2
-entry_4->
state 58: idle_1 idle_3 idle_4 token_in_4 wait_2
-ask_3->
state 59: idle_1 idle_4 token_in_4 wait_2 wait_3
-ask_1->
state 60: idle_4 token_in_4 wait_1 wait_2 wait_3
-transmit_4->
state 61: idle_4 token_out_4 wait_1 wait_2 wait_3
-exit_4->
state 62: after_4 idle_4 wait_1 wait_2 wait_3
-ask_4->
state 63: after_4 wait_1 wait_2 wait_3 wait_4
-entry_1->
state 64: token_in_1 wait_1 wait_2 wait_3 wait_4
-work_1->
state 65: cs_1 wait_2 wait_3 wait_4
-release_1->
state 66: idle_1 token_out_1 wait_2 wait_3 wait_4
-exit_1->
state 67: after_1 idle_1 wait_2 wait_3 wait_4
-entry_2->
state 68: idle_1 token_in_2 wait_2 wait_3 wait_4
-transmit_2->
state 69: idle_1 token_out_2 wait_2 wait_3 wait_4
-exit_2->
state 70: after_2 idle_1 wait_2 wait_3 wait_4
-entry_3->
state 71: idle_1 token_in_3 wait_2 wait_3 wait_4
-transmit_3->
state 72: idle_1 token_out_3 wait_2 wait_3 wait_4
-exit_3->
state 73: after_3 idle_1 wait_2 wait_3 wait_4
-entry_4->
state 74: idle_1 token_in_4 wait_2 wait_3 wait_4
-work_4->
state 75: cs_4 idle_1 wait_2 wait_3
-release_4->
state 76: idle_1 idle_4 token_out_4 wait_2 wait_3
-exit_4->
state 77: after_4 idle_1 idle_4 wait_2 wait_3
-entry_1->
state 78: idle_1 idle_4 token_in_1 wait_2 wait_3
-transmit_1->
```

```
state 79: idle_1 idle_4 token_out_1 wait_2 wait_3
-exit_1->
state 80: after_1 idle_1 idle_4 wait_2 wait_3
-entry_2->
state 81: idle_1 idle_4 token_in_2 wait_2 wait_3
-work_2->
state 82: cs_2 idle_1 idle_4 wait_3
-release_2->
state 83: idle_1 idle_2 idle_4 token_out_2 wait_3
-exit_2->
state 84: after_2 idle_1 idle_2 idle_4 wait_3
-entry_3->
state 85: idle_1 idle_2 idle_4 token_in_3 wait_3
-work_3->
state 86: cs_3 idle_1 idle_2 idle_4
-release_3->
state 87: idle_1 idle_2 idle_3 idle_4 token_out_3
-exit_3->
state 88: after_3 idle_1 idle_2 idle_3 idle_4
-entry_4->
state 89: idle_1 idle_2 idle_3 idle_4 token_in_4
-ask_3->
state 90: idle_1 idle_2 idle_4 token_in_4 wait_3
-transmit_4->
state 91: idle_1 idle_2 idle_4 token_out_4 wait_3
-exit_4->
state 92: after_4 idle_1 idle_2 idle_4 wait_3
-entry_1->
state 93: idle_1 idle_2 idle_4 token_in_1 wait_3
-ask_4->
state 94: idle_1 idle_2 token_in_1 wait_3 wait_4
-transmit_1->
state 95: idle_1 idle_2 token_out_1 wait_3 wait_4
-exit_1->
state 96: after_1 idle_1 idle_2 wait_3 wait_4
-entry_2->
state 97: idle_1 idle_2 token_in_2 wait_3 wait_4
-ask_1->
state 98: idle_2 token_in_2 wait_1 wait_3 wait_4
-transmit_2->
state 99: idle_2 token_out_2 wait_1 wait_3 wait_4
-exit_2->
state 100: after_2 idle_2 wait_1 wait_3 wait_4
-entry_3->
state 101: idle_2 token_in_3 wait_1 wait_3 wait_4
-work_3->
state 102: cs_3 idle_2 wait_1 wait_4
-release_3->
```

```
  state 103: idle_2 idle_3 token_out_3 wait_1 wait_4
  -exit_3->
  state 104: after_3 idle_2 idle_3 wait_1 wait_4
  -entry_4->
  state 105: idle_2 idle_3 token_in_4 wait_1 wait_4
  -work_4->
  state 106: cs_4 idle_2 idle_3 wait_1
  -ask_2->
  state 107: cs_4 idle_3 wait_1 wait_2
  -release_4->
  state 108: idle_3 idle_4 token_out_4 wait_1 wait_2
  -exit_4->
  state 109: after_4 idle_3 idle_4 wait_1 wait_2
  -ask_4->
  state 110: after_4 idle_3 wait_1 wait_2 wait_4
  -entry_1->
  state 111: idle_3 token_in_1 wait_1 wait_2 wait_4
  -transmit_1->
  state 113: idle_3 token_out_1 wait_1 wait_2 wait_4
  -exit_1->
  state 114: after_1 idle_3 wait_1 wait_2 wait_4
  -entry_2->
  state 115: idle_3 token_in_2 wait_1 wait_2 wait_4
  -work_2->
  state 116: cs_2 idle_3 wait_1 wait_4
  -release_2->
  state 117: idle_2 idle_3 token_out_2 wait_1 wait_4
  -exit_2->
  state 118: after_2 idle_2 idle_3 wait_1 wait_4
  -entry_3->
  state 119: idle_2 idle_3 token_in_3 wait_1 wait_4
  -ask_2->
  state 120: idle_3 token_in_3 wait_1 wait_2 wait_4
  -transmit_3->
  state 121: idle_3 token_out_3 wait_1 wait_2 wait_4
  -exit_3->
  state 122: after_3 idle_3 wait_1 wait_2 wait_4
  -entry_4->
  state 123: idle_3 token_in_4 wait_1 wait_2 wait_4
  -transmit_4->
  state 124: idle_3 token_out_4 wait_1 wait_2 wait_4
  -ask_3->
  state 125: token_out_4 wait_1 wait_2 wait_3 wait_4
  -exit_4->
* [accepting] state 126: after_4 wait_1 wait_2 wait_3 wait_4
  -entry_1->
  state 127: token_in_1 wait_1 wait_2 wait_3 wait_4
  -transmit_1->
```

```
  state 128: token_out_1 wait_1 wait_2 wait_3 wait_4
  -exit_1->
  state 129: after_1 wait_1 wait_2 wait_3 wait_4
  -entry_2->
  state 130: token_in_2 wait_1 wait_2 wait_3 wait_4
  -transmit_2->
  state 216: token_out_2 wait_1 wait_2 wait_3 wait_4
  -exit_2->
  state 165: after_2 wait_1 wait_2 wait_3 wait_4
  -entry_3->
  state 166: token_in_3 wait_1 wait_2 wait_3 wait_4
  -transmit_3->
  state 168: token_out_3 wait_1 wait_2 wait_3 wait_4
  -exit_3->
  state 169: after_3 wait_1 wait_2 wait_3 wait_4
  -entry_4->
  state 170: token_in_4 wait_1 wait_2 wait_3 wait_4
  -transmit_4->
  state 199: token_out_4 wait_1 wait_2 wait_3 wait_4
  -exit_4->
  state 126: after_4 wait_1 wait_2 wait_3 wait_4
0.040s

- ^D
bye
```

Appendix E

Options de l'outil *Mc*

```
Survival kit for the LTL model checker.

Command line (result of  mc -h):
================================

usage: mc [-h | -help]
       mc ktzfile [-f formula | formulafile] [-prelude file]
                  [-q | -v] [-b | -c | -p | -s | -g] [-wp n] [outfile]

 FLAGS         WHAT                                          DEFAULT
 -h | -help    this mode
 ktzfile       transition system input file
 -f formula    pass formula to be checked as a string
 formulafile   pass formula as a file (stdin if absent or -) -
 -prelude file loads command file on entry
 outfile       result file or -                              -
 output mode:
 -b            just prints truth value                       -c
 -c            prints summary of counter example             -c
 -p            prints counter example in full                -c
 -s            prints counter example loadable in stepper    -c
 -g            builds and prints full synchronized graph     -c
 information flag:
 -q            no banner nor times printed                   -v
 -v            prints banner and exec times for commands     -v
 ktz options:
 -wp (0|1|2)   (remove|preserve|force) wait properties       -wp 1


Interactive usage:
==================

One enters in interactive mode when no formulafile is specified.
```

```
1. Lexical matters:
-------------------

- A identifier is either:

Any place or transition identifier allowed in .net or .ndr descriptions, that is:

    any series of letters, digits, underscores "_" and primes "'"

    any sequence of characters enclosed in braces ({ ... }), in which "{", "}"
    (except the outer ones) and "\" are prefixed  by "\"

Any sequence of symbols from list ~,`,!,@,#,$,%,^,&,*,-,+,=,:,?,|,\,/,,,<,>,[,];

A qualified identifier: an identifier prefixed by either A. or L.

e.g.   hello, _p4'_, 123, >=<, or {variable x45}, are legal identifiers.

- The commands are: op, infix, prefix, forget, verb, output, source

Command names are not legal names for user defined operators or variables.

- When  analyzing  identifiers,  the  scanner  advances  as  right  as
possible.  So, in a juxtaposition  of identifiers, two symbolic or two
alphanumeric unbraced identifiers,  or e.g. an alphanumeric identifier
and a  command name, must  be separated by  a space.  But no  space is
necessary  between  identifiers  of  different  kinds  or  between  a
parenthesis (or ";") and an identifier.

- justaposition bind  tighter than infixes, and  all infixes associate
to the right. That is (f being a 3-ary operator):

       [] p1 => <> - p3 /\ f u v w
  is parsed as ([] p1) => ((<> (- p3)) /\ (f u v wp))

f - <> p1 (f p0 p1 (p4 /\ p5)) \/ f u v w
  is parsed as (f (- <> p1) p3 (f p0 p1 (p4 /\ p5)) \/ (f u v w)

  and f - <> p1 f p0 p1 p4 \/ f u v w
  is parsed as (f (- <> p1) f p0 p1 p4) \/ (f u v w)
  which is ill-typed

- infixes are associated precedence in 0..3 (see below). Infixes with higher
precedence bind tighter than those with lower precedence.


2. The initial environment:
---------------------------
```

```
It is made up of (pushed in that order):

- The atomic state and event propositions. They have the names
  captured in the .ktz file, i.e. those of the places and transitions
  of the Petri net if the .ktz file was generated by tina.

- Then, the logic and arithmetic primitives, constituted of:

   constants:   T (true), F (false),
  dead (deadlock property), div (temporal divergence property)
   prefixes:   [] (always), <> (eventually), () (next), - (negation)
   infixes:   /\ (and), \/ (or), => (implies), <=> (equivalent),
  U (until), V (release), of precedence 0
    <=, >=, =, of precedence 1,
+, of precedence 2
   *, of precedence 3

- Then the user defined operators.

Since the syntactic classes of atomic proposition, logic primitives,
integer. and user defined operators, overlap, we must have some way of
disambiguating identifiers. For this:

- unqualified identifiers are bound to the last pushed environment entry with that name;

- identifiers qualified by A (e.g. A.p1) are bound to the atomic prop with that
  name with the qualifier removed (e.g. p1);

- identifiers qualified by L (e.g. L./\) are bound to the logic primitive with that
  name with the qualifier removed (e.g. /\);

- integers are analyzed as integers, except if prefixed by A. , in which case
  they are taken as the atomic property with that name.


3. Fixity:
----------

Identifiers declared infix (binary logic primitives or user defined
operators declared by "infix") must be used in infix notation;

Identifiers declared prefix (unary logic primitives or user defined
operators declared by "prefix") must be used in prefix notation (in a
juxtaposition of identifiers, prefix operators associate with the
right expression);

User defined operators or primitives accessed by their qualified names
```

```
(e.g. L./\), must be used  in functional notation.  E.g. an operator f
of  arity n  is  invoked by  f  a1 ...  an, with  the  ai enclosed  in
parentheses if necessary;


4. Commands and effects: ("exp" is any ltl expression, x, y, f, xi are identifiers)
------------------------

In interactive  mode, all commands  must terminate with ";".  In batch
mode, the final  ";" may be omitted (e.g. in a  formula file passed by
the command line, or in sourced files). The commands accepted are:

exp;
evaluates LTL expression exp;

op f x1 ... xn = exp;

   declares an operator f or arity n (n >= 0);

infix [n] x f y = exp;

      declares a binary operator f in infix notation. n is an
      optional integer in 0..3 specifying precedence.

prefix f x = exp;

       declares a unary operator f in prefix notation;

forget f1 ... fn;

       Removed items names f1 ... fn from the environment, and their fixity information;

       note:  does  not remove  them  from  structures  A and  L,  so
       forgotten atomic propositions or logic primitives can still be
       accessed by their qualified names;

source [file | "file"];

       reads at toplevel  the contents of file. The  file name can be
       optionally surrounded  by string quotes (this  is necessary if
       the name includes spaces);

verb [true | false | debug];

     verbosity level. Can also be set by the command line by flags -v | -q;

     verb true (default) prints the banner, prompts, results of commands, and evaluation time
     verb false just prints the results of evaluation of LTL expressions (useful in batch mod
```

```
     verb debug prints extra information (useful to developpers);

output [proof | fullproof | stepper | graph | quiet]

       specify   effects   and   results   of   evaluations   of   LTL
       expressions. Can also be set  by the command line by flags -c,
       -s, -g, -b;

       output proof (default): evaluations return TRUE, or FALSE with a counter example in sh
       output proof (default): evaluations return TRUE, or FALSE with a full counter example;
       output stepper: as output proof except the counter example is in stepper format;
       output graph: not implemented yet
       output quiet: evaluations just return TRUE or FALSE, without counter examples.
```

List of Figures

Bibliography

[ABB+02] S. Abdellatif, B. Berthomieu, J.P. Bodeveix, J.M. Farines, M. Filali, C. Lohr, P. Michel, O. Nasr, G. Padiou, P.O. Ribet, S. Sarpag, and F. Vernadat. Rapport sur les sémantiques, concepts, techniques et outils. Technical report, COTRE Project - LAAS Report No 02384, 2002. 75 pages.

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of* 5^{th} *IEEE Symp. on Logic in Computer Science*, 1990.

[ARI97] Airlines Electronic Engineering Committee, 2551, Riva road, Annapolis, Maryland 21401. *Avionics Application Software Standard Interface, ARINC Specification 653*, january 1997.

[Arn87] A. Arnold. *Système de transitions finis et sémantique des processus communicants.* Masson, 1987.

[BBD01] B. Berthomieu, M. Boyer, and M. Diaz. *Les Réseaux de Petri. Modèles fondamentaux*, chapter Les réseaux de Petri temporels, pages 161–199. M. Diaz. Hermes Science, Traité IC2 Information-Commande-Communication, N ISBN 2-7462-0250-6, 2001.

[BBD+03a] B. Berthomieu, J.P. Bodeveix, S. Devulder, J.M. Farines, M. Filali, J.L. Lambert, P. Michel, P.O. Ribet, and F. Vernadat. Choix des techniques et outils de vérification. Technical report, COTRE Project - LAAS Report 03320, 2003. 22 pages.

[BBD+03b] B. Berthomieu, J.P. Bodeveix, S. Devulder, J.M. Farines, M. Filali, J.L. Lambert, P. Michel, P.O. Ribet, and F. Vernadat. Expérimentation sur l'étude de cas : validation de fragments. Technical report, COTRE Project - LAAS Report 03277, 2003. 32 pages.

[BBD+03c] B. Berthomieu, J.P. Bodeveix, S. Devulder, J.M. Farines, M. Filali, J.L. Lambert, P. Michel, P.O. Ribet, and F. Vernadat. Présentation des techniques et outils de vérification. Technical report, COTRE Project - LAAS Report, 2003. 47 pages.

[BBD+04] B. Berthomieu, J.P. Bodeveix, S. Devulder, J.M. Farines, M. Filali, J.L. Lambert, P. Michel, O. Nasr, G. Padiou, P.O. Ribet, and F. Vernadat. Traductions de V-Cotre vers les formalismes temporels et temorisés. Technical report, COTRE Project - LAAS Report 04371, 2004. 51 pages.

[BBF+04] B. Berthomieu, J.P. Bodeveix, J.M. Farines, M. Filali, P. Gaufillet, J.L. Lambert, P. Michel, O Nasr, G. Padiou, P.O. Ribet, and F. Vernadat. Cotre : rapport de prospective. Technical report, COTRE Project - LAAS Report 04372, 2004. 20 pages.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking : 10^{20} states and beyond. In *Proceedings of* 5^{th} *IEEE Symp. on Logic in Computer Science*, 1990.

[BD91] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 1991.

[Ber85] G. Berthelot. Checking properties of nets using transformations. In *Advances in Petri Nets*, LNCS 222, pages 19–40, 1985.

[BF99] Béatrice Bérard and Laurent Fribourg. Reachability analysis of (timed) Petri nets using real arithmetic. In *CONCUR'99*, pages 178–193. LNCS, 1999.

[BF00] J-P. Bodeveix and M. Filali. Expérimentation de méthodes d'accélération. In *Proceedings of Formalisation des Activités Concurrentes (FAC'00)*, 2000.

[BRV03a] 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. In *Hermes ISBN 2-7462-0778-8*, 2003.

[BRV+03b] B. Berthomieu, P.O. Ribet, F. Vernadat, J. Bernartt, J.M. Farines, J.P. Bodeveix, M. filali, G. Padiou, P. Michel, P. Farail, P. Gauffilet, P. Dissaux, and J.L. Lambert. Towards the verification of real-time systems in avionics : the cotre approach. In *Workshop on Formal Methods for Industrial Critical Systems (FMICS'2003)*, pages 201–216, 2003.

[BRV04] B. Berthomieu, P.O. Ribet, and F. Vernadat. The tool TINA - construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research (IJPR)*, 2004.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8) :677–691, August 1986.

[BV03] B. Berthomieu and F. Vernadat. State class constructions for branching analysis of time Petri nets. In *Proceedings of TACAS'03*. Springer Verlag, LNCS 2619, 2003.

[CCO+04] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *IFM, LNCS 2999*, 2004.

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

[CGP00] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. Hardcover, 2000.

[Cor96] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on software engineering*, VOL. 22(NO. 3), 1996.

[CX97] F. Chu and X.Xie. Deadlock analysis of Petri nets using siphons and mathematical programming. In *IEEE Trans. on Robotics and Automation*, volume 13-6, pages 793–804, 1997.

[ES01] J. Esparza and C. Schroter. Net reductions for LTL Model-Checking. In *Correct Hardware Design and Verification Methods (CHARME'01)*. Springer-Verlag, LNCS 2144, 2001.

[FM91] Jean-Claude Fernandez and Laurent Mounier. A toolset for deciding behavioral equivalences. In *CONCUR'91*. Springer Verlag, LNCS 527, 1991.

[Gen87] J. Genrich. Predicate/transition nets. In *Petri Nets : Applications and relationships to other models of concurrency, Part I, LNCS 254*, 1987.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme : An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, 1997.

[GO01] P. Gastin and D. Oddoux. Fast LTL to buchi automata translation. In *Proceedings of Computer Aided Verification (CAV'01)*, pages 53–65. Springer Verlag, LNCS 2102, 2001.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of CAV'90*, pages 321–340. ACM, DIMACS volume 3, 1990.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems.* PhD Thesis. Springer Verlag, LNCS 1032, 1996.

[GW93] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2) :149–164, 1993.

[Had01] S. Haddad. *Les Réseaux de Petri. Modèles fondamentaux*, chapter Décidabilité et complexité de problèmes de réseaux de Petri, pages 119–158. M. Diaz. Hermes Science, Traité IC2 Information-Commande-Communication, N ISBN 2-7462-0250-6, 2001.

[HMP91] T.A. Henziger, Z. Manna, and A. Pnueli. Timed transition systems. In *REX Workshop - LNCS 600*, pages 226–251, 1991.

[HV01] S. Haddad and F. Vernadat. *Les Réseaux de Petri. Modèles fondamentaux*, chapter Méthodes d'analyse des réseaux de Petri, pages 69–117. M. Diaz. Hermes Science, Traité IC2 Information-Commande-Communication, N ISBN 2-7462-0250-6, 2001.

[II83] M. Itoh and H. Ichikawa. Protocol verification using reduced reachability analysis. In *Proceedings of IECE vol. E66, no. 2 pp 88-93*, 1983.

[Jen86] K. Jensen. Coloured petri nets. In *Petri Nets : Central Model and Their Properties*, pages 248–299. Springer-Verlag, LNCS 254, 1986.

[KM69] R. Karp and R. Miller. Parallel program schemata. *Computer and System Sciences 3*, 1969.

[Kri71] S.A Kripke. Semantical considerations on modal logic. In *Reference and Modality.* L. Linsky, Ed London : Oxford University Press, 1971.

[KT00] B. Karacali and K.C. Tai. Model checking based on simultaneous reachability analysis. In *Proceedings of SPIN Workshop'00.* Springer Verlag, LNCS 1885, 2000.

[Maz86] A. Mazurkiewicz. Trace theory. In *Petri Nets : Applications and Relationships to Other Model of Concurrency, Advances in Petri nets 1986, Part II; Proceedings of an advanced Course*, pages 279–324. Springer Verlag, LNCS 255, 1986.

[MF76] P.M. Merlin and D.J. Farber. Recoverability of communication protocols, implication of a theoritical approach. *IEEE Transactions on Communications*, Septembre 1976.

[Mil80] R. Milner. *A Calculus of communicating Systems*, chapter A case study in synchronisation, and proof techniques, pages 32–46. LNCS 92. Springer-Verlag, 1980.

[Mil85] R. Milner. *Communication and Concurrency.* Prentice Hall, 1985.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MPR03] F. Magniette, L. Pilard, and B. Rozoy. Model-checking et produit synchronisé. In *MSR*, 2003.

[MT00] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, 2000.

[MV98] F. Michel and F. Vernadat. Maîtrise de l'explosion combinatoire réduction du graphe de comportement. *Technique et Science Informatiques*, Volume 17(n 7), 1998.

[NT87] M. Naimi and M. Trehel. An improvement of the $\log N$ distributed algorithm for mutual exclusion. In *International Conference on Distributed Computing Systems*, pages 371–377. IEEE Computer Society Press, 1987.

[NTA96] M. Naimi, M. Trehel, and A. Arnold. A $\log N$ distributed mutual exclusion algorithm based on path reversal. *Parallel and distributed computing*, 34 :1–13, 1996.

[Ove81] Overman. *Verification of concurrent systems : function and timing.* PhD thesis, University of California, 1981.

[Pag97] F. Pagani. Ordres partiels pour la vérification de systèmes temps réel. In *Thèse de l'école nationale supérieure de l'aéronautique et de l'espace*, 1997.

[PCB98] D. Paun, M. Chechik, and B. Biechele. Production cell revisited. In *SPIN'98*, 1998.

[Pel93] D. Peled. All from one, one for all : On model checking using representatives. In *Proceedings of CAV'93*, pages 409–423. Springer Verlag, LNCS 697, 1993.

[Pel98] D. Peled. Ten years of partial order reduction. In *Proceedings of CAV.* Springer Verlag, LNCS 1427, 1998.

[PF90] D. H. Pitt and D. Freestone. The derivation of conformance tests from lotos specifications. *IEEE Transactions on Software Engineering*, 1990.

[PL00] Paul Pettersson and Kim G. Larsen. Uppaal2k. *European Association for Theoretical Computer Science*, 70, 2000.

[Pnu86] A. Pnueli. Specification and developement of reactive system. *Information Processing*, pages 845–858, 1986.

[Ray91] M. Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Oper. Syst. Rev.*, 25(2) :47–50, 1991.

[Rei85] W. Reisig. *Petri Nets : an Introduction.* Springer-Verlag,EATCS, 1985.

[RVB02] P.O. Ribet, F. Vernadat, and B. Berthomieu. On combining the persistent sets method with the covering steps graph method. In *FORTE 2002*, pages 344–359. Springer Verlag, LNCS 2529, 2002.

[SAE02] SAE. Aerospace information report. Avionics Architecture Description Language. Technical Report AS5506, SAE, march 2002.

[TY96] S. Tripakis and S. Yovine. Analysis of timed systems based on time-abstracting bisimulations. In *Proceedings of Computer Aided Verification (CAV'96).* Springer Verlag, LNCS 1102, 1996.

[Val88a] A. Valmari. Error detection by reduced reachability graph generation. In *Proceedings of ATPN'88.* Springer Verlag, LNCS 424, 1988.

[Val88b] A. Valmari. *State Space Generation : Efficiency and Practicality.* PhD thesis, Tampere University of Technology, 1988.

[Val89] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of ATPN'89.* Springer Verlag, LNCS 483, 1989.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Proceedings of CAV'90*, pages 25–42. ACM, DIMACS volume 3, 1990.

[Val93] A. Valmari. On-the-fly verification with stubborn sets. In *Proceedings of CAV'93*, pages 397–408. Springer Verlag, LNCS 697, 1993.

[Val98] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I : Basic Models, Advances in Petri Nets*, pages 429–528. Springer-Verlag, 1998.

[VAM96] F. Vernadat, P. Azéma, and F. Michel. Covering step graph. In *Proceedings of ATPN'96*. Springer Verlag, LNCS 1091, 1996.

[VM97] F. Vernadat and F. Michel. Covering step graph preserving failure semantics. In *Proceedings of ATPN'97*. Springer Verlag, LNCS 1248, 1997.

[Wes86] C.H. West. Protocol verification by random state exploration. In *PSTV VI*, pages 233–242, 1986.

[WG93] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proceedings of CONCUR'93*. Springer Verlag, LNCS 575, 1993.

[Yov97] S. Yovine. Kronos : A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1), 1997.

[YS96] T. Yoneda and B-H. Schlingloff. Efficient verification of parallel real-time systems. *Journal of Frmal Methods in System Design*, 1996.

[ZDD93] M.C. Zhou, F. Dicesare, and A.A. Desrochers. A hybrid methodology for synthesis of petri net models for manufacturing systems. In *IEEE Trans. on Robotics and Automation 8 :3*, pages 350–361, 1993.

Thèse de Pierre-Olivier Ribet

Vérification formelle de systèmes.
Contribution à la réduction de l'explosion combinatoire.

RÉSUMÉ

La vérification formelle de systèmes concurrents temps réels se heurte au problème de l'explosion du nombre d'états à explorer. Ce problème connu sous le nom « d'explosion combinatoire » à plusieurs causes. Cette thèse s'intéresse à deux d'entre-elles.

- Pour lutter contre l'explosion due à la représentation du parallélisme par l'entrelacement d'actions, cette thèse propose des techniques basées sur l'approche des ordres-partiels pour construire un graphe réduit. Pour exploiter les ordres-partiels, les techniques proposées utilisent la construction de « pas de transitions » afin de limiter le nombre d'états explorés. Différentes constructions des « pas de transitions » sont proposées en fonction de la classe de propriétés que l'on souhaite préserver (Blocages, Équivalence de traces, LTL_{-X}).
- Pour lutter contre l'explosion due aux contraintes temporelles, cette thèse propose une approche par sur-approximation du comportement. L'objectif est d'avoir un graphe abstrait du comportement de la sur-approximation plus petit que celui du système. Comme classiquement, les techniques d'abstractions permettent d'obtenir une procédure de décision semi-effective. Lorsque l'analyse de la sur-approximation ne permet pas de conclure, la thèse propose une méthode effective permettant de conclure pour les formules de LTL : le système est analysé, guidé par les résultats obtenus sur la sur-approximation.

Cette thèse présente les algorithmes de ces différentes techniques de réduction et l'outil tina (http ://www.laas.fr/tina) dans lequel ils ont été implémentés.

MOTS CLÉS

systèmes concurrents, vérification formelle, explosion combinatoire, systèmes temps-réel

---

Formal verification of systems.
Contribution to the reduction of Combinatorial Explosion.

ABSTRACT

Formal verification technique often runs up against the combinatorial explosion problem : the number of states of the transitions system grows in an exponential way with respect to the component count of the system. This thesis study two causes :

- For untimed systems, concurrency is represented by actions interleaving which produces a lot of combinations. This thesis investigates partial-order approaches to compute a reduced graph. Techniques based on « transitions step » are proposed for different properties classes.
- For timed systems, time constraints can produce explosion of the exploration. This thesis proposes to compute an over-approximation of the system, in order to build a smaller behavior graph. The property is checked on the over-approximation. If the analysis of the over-approximation doesn't allow to conclude, the system behavior is checked, guided by results founded on the over-approximation.

KEWORDS

concurrent systems, formal verification, combinatorial explosion, real time systems