Contribution à l'algorithmique de la vérification

(Mémoire d'habilitation à diriger des recherches)

Jean-Michel COUVREUR

Laboratoire Bordelais de Recherche en Informatique
CNRS UMR 5800 - Université Bordeaux I

Laboratoire Spécification et Vérification
CNRS UMR 8643 - ENS de Cachan

Présenté le 6 juillet 2004 à l'ENS de Cachan devant un jury composé de :

– André Arnold	examinateur
– Javier Esparza	examinateur
– Alain Finkel	examinateur
– Paul Gastin	examinateur
– Serge Haddad	rapporteur
– Philippe Schnoebelen	examinateur
– Igor Walukiewicz	rapporteur
– Pierre Wolper	rapporteur

Table des matières

Chapitre 1

Introduction

Les techniques de vérifications ont déjà plus de vingt ans. Elles ont été développées dans un premier temps par des équipes de chercheurs et sont de plus en plus utilisées dans le milieu industriel pour l'analyse d'une grande variété de systèmes (systèmes matériels, logiciels, systèmes réactifs, systèmes temps réel). Il est maintenant prouvé que ces techniques sont efficaces et sont fréquemment utilisées pour détecter des bogues dans des cas industriels. De nombreuses études sont en cours pour élargir leurs champs d'applications et améliorer leur efficacité. Ceci nous conduit à penser que les applications industrielles vont se multiplier de manière significative dans les prochaines années.

Ce mémoire présente mes travaux les plus récents, consacrés à la vérification de systèmes. J'ai concentré mes efforts sur l'amélioration de différents aspects de l'algorithmique de la vérification : la logique temporelle linéaire (LTL), la vérification par ordre partiel et la vérification symbolique. Mes recherches sont en partie alimentées par des préoccupations concrètes abordées dans le cadre de projets industriels ou des projets universitaires. Par exemple, dans le cadre du projet CLOVIS (un projet de recherche exploratoire DGA), j'ai élaboré et implémenté une structure originale de diagrammes de décision adaptée à la vérification symbolique et nous l'avons appliquée à la vérification de programmes VHDL. Mes autres recherches sont beaucoup plus académiques, comme la vérification de formules LTL ou la vérification par ordre partiel.

J'ai organisé cette présentation en trois chapitres :

1. Le premier chapitre est dédié à mes contributions concernant la logique temporelle linéaire [14, 15, 22]. Nous consacrons une grande part au problème de la construction de l'automate d'une formule LTL. Il contient une discussion sur le type d'automates à produire, plusieurs descriptions de constructions globales [15] et locales [14], ainsi qu'une étude expérimentale. Nous présentons ensuite notre algorithme de vérification à la volée [14]. Nous terminons ce chapitre par nos récents travaux sur la vérification de systèmes probabilistes [22].
2. Dans le deuxième chapitre, nous décrivons nos contributions, tirées de [20, 21, 18, 19, 23], sur une méthode basée sur l'ordre partiel : le dépliage de réseaux de Petri. Nous donnons les éléments théoriques conduisant à la

notion de dépliage de réseaux de Petri, ainsi que les définitions de préfixes finis à partir desquelles des algorithmes de vérification de propriétés de sûreté sont développés. Nous décrivons de manière synthétique nos travaux sur les graphes de préfixes finis [20, 21, 18, 23] et leurs applications à la vérification de propriétés de logique temporelle. Nous terminons ce chapitre par une étude du dépliage de réseaux de Petri symétriques et du produit de réseaux de Petri [19].

3. Le troisième chapitre concerne nos travaux sur les méthodes de vérification symbolique [17, 7, 16]. Nous présentons une structure de donnée à la BDD, les diagrammes de décisions de données (DDD) [17], élaborée pour traiter des systèmes décrits dans des langages de haut niveau. Nous montrons la puissance de description des DDD sur une étude de cas [7] : le BART de San Francisco. Nous terminons ce chapitre par nos travaux récents sur une représentation à la BDD des automates finis déterministes, les automates partagés.

Chapitre 2

Logique temporelle linéaire

La logique temporelle est un langage puissant permettant de décrire des propriétés de sûreté, d'équité et de vivacité de systèmes. Elle est utilisée comme langage de spécification dans des outils tel que SPIN [48] et SMV [58]. Cependant, vérifier qu'un système fini respecte une telle spécification est PSPACE-complet [75]. En pratique, les techniques de vérification sont confrontées à un problème d'explosion combinatoire du nombre d'états du système et de celui de l'automate codant la formule. De nombreuses techniques ont été élaborées pour faire face à ce problème d'explosion. Nous pouvons noter les techniques de vérification à la volée combinées avec des techniques de réduction à base d'*ordre partiel* (SPIN [48]). La représentation symbolique par les diagrammes de décisions permet de coder un système et l'automate d'une formule de manière concise et ainsi de repousser les limites de la vérification (SMV [58]). Chacune de ces méthodes ont leurs succès sur des systèmes industriels prouvant leur bien-fondé.

En 1999 [14], j'ai proposé de nouveaux algorithmes pour résoudre les deux problèmes clefs de la vérification à la volée d'une formule LTL :

- Construire à la demande un automate représentant une formule LTL ;
- Tester à la volée si l'automate résultant du produit synchronisé du système et de l'automate de la propriété est vide.

Ces deux problèmes avaient déjà été résolus. L'algorithme de construction d'automates proposé dans [37] produit non seulement des automates de tailles raisonnables, mais opère aussi à la demande. Les algorithmes proposés dans [46, 12, 40] testent à la volée le problème du vide d'un automate. Cependant l'enseignement de ces techniques m'ont conduit à les remettre en question. D'une part, l'algorithme de construction [37] produit pour certaines formules simples des automates compliqués et certains aspects de la construction défiaient l'intuition. D'autre part, pourquoi aucun des algorithmes pour tester le vide n'était basé sur le fameux algorithme de Tarjan [77]. Mon objectif initial était donc pédagogique : (1) développer une technique de construction d'automates facile à comprendre et construisant de petits automates pour de petites formules ; (2) reformuler l'algorithme de Tarjan pour résoudre le problème du vide d'un automate.

J'ai conçu une construction d'automates qui est dans quasiment tous les cas

meilleure que celle proposée dans [37] : elle produit des automates plus petits. L'algorithme construit une variante des automates de Büchi : des automates à transitions. Contrairement aux automates de Büchi, les conditions d'acceptation portent ici sur les transitions. Ma construction est très voisine de celle proposée dans [37]. Elle est basée sur une technique de tableaux [88, 89]. Le point clef de ma méthode est l'utilisation de calcul symbolique, qui permet de simplifier les expressions de manière naturelle et ainsi de réduire le nombre d'états. De plus, une implémentation simple et efficace peut être obtenue en utilisant des diagrammes de décisions binaires.

Mon algorithme de vérification est une variation simple de l'algorithme de Tarjan. Il a les caractéristiques suivantes :

- L'algorithme est conçu pour fonctionner à la volée, c-à-d, durant le parcours de l'automate produit, un comportement accepté est détecté dès que le graphe parcouru en contient un.
- L'algorithme traite directement des automates à transitions, c-à-d, il n'est pas nécessaire de transformer l'automate en un simple automate de Büchi.

Les algorithmes initialement proposés [46, 12, 40] n'ont aucune de ces bonnes propriétés.

En 2003 [22], je me suis intéressé à la vérification de formules LTL sur des systèmes probabilistes. Les algorithmes probabilistes sont conçus dans de nombreux domaines de l'informatique, et plus particulièrement en algorithmique distribuée. En fait, dans ce domaine, il existe des problèmes qui n'admettent pas de solutions algorithmiques déterministes satisfaisantes. Ainsi, le choix de mettre en œuvre des solutions probabilistes devient nécessaire. L'extension et la conception de technique de vérification efficace pour les systèmes concurrents probabilistes constituent encore à ce jour un chalenge.

Le principal résultat de notre travail [22] est la conception d'une méthode à base d'automates pour la vérification de formules LTL sur des systèmes probabilistes. Le point essentiel est que l'algorithme proposé résout le problème avec une complexité optimale [13]. Comme dans le cas non probabiliste, nous synchronisons le système avec l'automate de la négation de la formule. Cependant, nous utilisons une construction particulière des automates de la formule. Cette construction que j'ai proposée dans [15], produit des automates ayant des propriétés spécifiques. Nous avons ainsi exploité ces bonnes propriétés pour éviter la déterminisation de ces automates qui est une étape coûteuse conduisant à une complexité doublement exponentielle en temps [84].

Ce chapitre est organisé en 4 parties. La première partie se veut informelle ; elle pose les définitions relatives aux modèles du temps linéaires et décrit les principes de la vérification à base d'automates. La deuxième partie aborde le problème de la traduction d'une formule LTL en un automate. Cette partie contient une discussion sur le type d'automates à produire, donne deux méthodes de construction, et termine par une étude expérimentale comparative avec des méthodes existantes. La troisième partie décrit une adaptation de l'algorithme de Tarjan pour le test du vide d'un automate. La quatrième partie est consacrée à la vérification d'une formule LTL sur un système probabiliste.

2.1 Modèles du temps linéaire

Cette section décrit de manière informelle les modèles du temps linéaire : les comportements linéaires, les systèmes de transitions étiquetés et la logique temporelle linéaire. Nous terminons par la description des principes de la vérification à la base d'automates.

Comportements linéaires. Dans les logiques du temps linéaire, un comportement est modélisé par une suite infinie d'événements, lors desquels il est possible d'observer les états et des actions d'un système. L'ensemble de ces observations est appelées propositions élémentaires. Formellement un comportement linéaire sur un ensemble de propositions élémentaires AP est une structure $r = (T, \sigma, \lambda)$ où T est un ensemble de transitions, σ un mot infini sur T et $\lambda : T \to 2^{AP}$ une fonction d'étiquetage. La trace d'un comportement linéaire r est le mot infini sur l'alphabet 2^{AP} défini par $Trace(r) = \lambda(\sigma)$.

Système de transitions étiquetées. Les modèles de systèmes ont une sémantique à base de systèmes de transitions. Un tel système est défini par un quintuple $M = (S, S_0, T, \alpha, \beta, L)$ où S est un ensemble d'états, S_0 un ensemble d'états initiaux, T un ensemble de transitions, $\alpha, \beta : T \to S$ sont les fonctions identifiant l'origine et le but des transitions, et $\lambda : T \to 2^{AP}$ est une fonction d'étiquetage des transitions.

Les comportements linéaires d'un système de transitions sont définis par les mots infinis de transitions décrivant des chemins valides du système démarrant d'un état de S_0. Dans la sémantique du temps linéaire, un système est identifié par l'ensemble des traces de ses comportements linéaires. Notez que les comportements finis terminant dans un état puit ne sont pas pris en compte. Sans perdre de généralité, ce défaut est résolu en ajoutant une transition bouclante à chaque état puit. Dans ce modèle, une propriété d'état est modélisée par une propriété de transition en affectant à chaque transition la valeur de vérité de son état source. Dans la suite, nous ne considérerons que des systèmes à nombre fini d'états et de transitions.

Logique temporelle linéaire. Cette logique permet d'énoncer des propriétés sur les traces des comportements d'un système. Elle permet de parler de l'évolution d'un système au cours du temps en examinant la suite des événements observés lorsque le système réalise un comportement linéaire. Les formules sont construites à partir des propriétés élémentaires, des opérateurs et constantes booléennes et de deux opérateurs temporels : l'opérateur unaire X (qui se lit Next) et l'opérateur binaire U (qui se lit Until). On définit la relation de satisfaction $\sigma, i \models f$ d'une formule f pour un mot infini de $\sigma = \sigma_0 \cdots \sigma_i \cdots$ de $(2^{AP})^\omega$ à l'instant i par induction sur la construction de la formule :

1. $\sigma, i \models p$ si $p \in \sigma_i$ pour tout $p \in AP$,
2. $\sigma, i \models \neg f$ si $\neg(\sigma, i \models f)$,
3. $\sigma, i \models f \wedge g$ si $(\sigma, i \models f) \wedge (\sigma, i \models g)$,
4. $\sigma, i \models Xf$ si $\sigma, i+1 \models f$,

5. $\sigma, i \models f\,U\,g$ si $\exists j : j \geq i \wedge \sigma, j \models g \wedge (\forall k : i \leq k < j \Rightarrow \sigma, k \models f)$.

L'interprétation intuitive de $X f$ et $f\,U\,g$ est la suivante : $X f$ est vraie à l'instant i si f l'est à l'instant suivant ; $f\,U\,g$ est vraie à l'instant i si g est vraie à un instant futur j et entre les instants i et j, f est toujours vraie. On utilise librement les abréviations logiques habituelles suivantes $\top$, $\bot$, $f \vee g$, $f \Rightarrow g$, et $f \Leftrightarrow g$, ainsi que $F f \stackrel{def}{\equiv} \top\,U f$, $G f \stackrel{def}{\equiv} \neg F \neg f$. $F f$ dit que f est vraie dans un instant futur, le dual $G f$ dit que f est vraie dans tous les instants futurs.

L'évaluation de relation $\sigma, 0 \models f$ à l'instant 0 définit la relation de satisfaction $\sigma \models f$ d'une formule pour une trace d'un comportement linéaire. Nous dirons qu'un système de transitions étiquetées vérifie une formule f, noté $M \models f$, si toutes les traces de ses comportements vérifient f.

Vérification par automates. Les principes de la vérification de systèmes finis par automates pour LTL sont dû à Vardi, Wolper [85]. Ils reposent sur le fait que l'ensemble des mots infinis satisfaisant une formule LTL forme un langage régulier et est représentable par un ω-automate. Pour appliquer cette propriété, un système est vu comme un générateur de mots infinis. Ainsi, si M est un système et f une formule, tester $M \models f$ se ramène à un problème purement sur les automates : tester si $M \cap \neg f$ est vide, c-a-d n'accepte pas de mot infini. Cette approche pose clairement les problèmes algorithmiques de la vérification : (1) calculer un ω-automate pour une formule LTL ; (2) calculer l'intersection de deux automates ; (3) tester le vide d'un automate. Dans la suite de notre présentation, nous examinerons nos contributions aux problèmes (1) et (3) et montrerons comment cette approche est appliquable à la vérification de systèmes probabilistes.

2.2 Traduction d'une formule LTL en ω-automate

De nombreux travaux traitent de la construction d'un automate pour une formule LTL. Nous ne donnerons pas une généalogie des méthodes. On pourra trouver dans [90, 25, 78] de très bons états de l'art. Mes contributions sur ce thème sont multiples. Premièrement, elles concernent le choix du type d'automates construits : les automates à transitions. Dans ces automates, les conditions d'acceptation portent sur les transitions infiniment rencontrées plutôt que sur les états. Cette idée a été reprise par des constructions récentes et donne à ces méthodes une efficacité quasiment immédiate. Deuxièmement, j'ai développé une méthode produisant des automates adaptés aux problèmes de la vérification probabiliste. Un aspect intéressant de cette traduction est la représentation directe de l'automate sous la forme d'un BDD. Troisièmement, j'ai développé une construction bien adaptée aux techniques de vérification à la volée : l'objectif est de construire localement la liste des transitions successeurs d'un état. Le point clef de cette technique est l'utilisation de BDD pour représenter et simplifier ces listes et ainsi réduire la taille de l'automate produit.

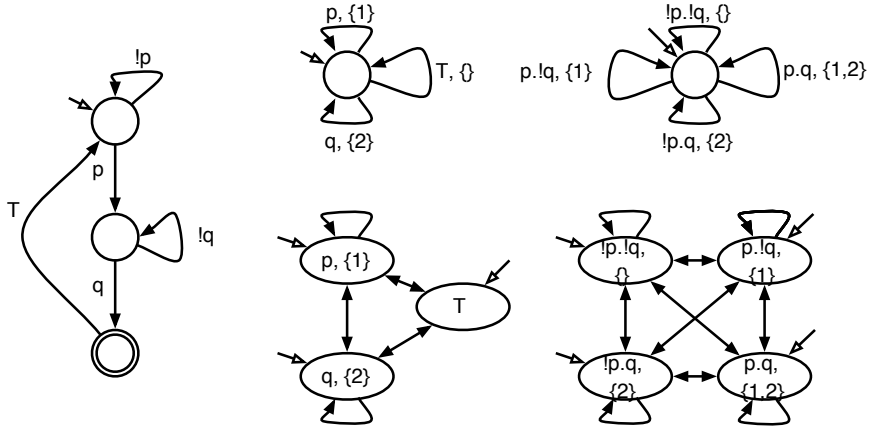


FIG. 2.1 – Automates pour la formule $GFp \wedge GFq$

2.2.1 Automates à transitions

Un automate reconnaissant des mots infinis est un système fini adjoint d'une condition définissant les chemins infinis pour être acceptés. Les mots infinis acceptés par un automate sont les traces des chemins infinis acceptés. Dans le cas des automates de Büchi, un chemin accepté doit traverser infiniment un ensemble d'états appelés états d'acceptation. Pour les automates que nous considérons, les automates à transitions, la condition d'acceptation porte sur les transitions parcourues infiniment. Dans notre étude, les conditions sont des conjonctions de conditions de Büchi. Une condition est définie par un ensemble d'ensembles de transitions. Un chemin est accepté s'il traverse infiniment tous les ensembles de la condition.

Définition 2.1 (Automate à transitions) *Soit AP un ensemble fini de propositions élémentaires. Un automate à transitions sur AP est une structure $A = (S, S_0, T, \alpha, \beta, \lambda, Acc)$ où*

- *S est un ensemble fini d'états,*
- *$S_0 \subseteq S$ est un ensemble d'états initiaux,*
- *T est un ensemble fini de transitions,*
- *$\alpha, \beta : T \to S$ définissent la source et la destination des transitions,*
- *$\lambda : T \to 2^{AP}$ est la fonction d'étiquetage des transitions,*
- *$Acc \subseteq 2^T$ est l'ensemble d'ensembles d'acceptation.*

Un chemin infini $\rho = t_0 \cdots t_n \cdots$ de l'automate est accepté si

- *$src(t_0) \in S_0$ et $\forall i \in \mathbb{N} : \beta(t_i) = \alpha(t_{i+1})$,*
- *$\forall acc \in Acc, \forall i \in \mathbb{N}, \exists j \in \mathbb{N} : j > i \wedge t_j \in acc$.*

Les mots infinis de $(2^{AP})^\omega$ acceptés par A sont les traces des chemins infinis acceptés par A. Notons $L(A)$, l'ensemble des mots acceptés par A et $L(A, s)$ l'ensemble des mots acceptés par l'automate A avec s comme unique état initial.

Plusieurs variantes d'automates ont été utilisées pour traduire des formules LTL. Traditionnellement, les conditions portent sur les états ($Acc \subseteq 2^S$). Dans certains travaux, les propriétés sont attribuées aux états ($\lambda : S \to 2^{AP}$). Les

propriétés étiquetant les états ou les transitions peuvent aussi prendre la forme d'expressions booléennes sur les propositions élémentaires ($\lambda : T \to 2^{2^{AP}}$ ou $\lambda : S \to 2^{2^{AP}}$). Notez que notre définition capture tous ces automates en reportant, si besoin, les propriétés et les conditions d'acceptation sur les transitions, et en développant les expressions booléennes étiquetant éventuellement les transitions. Nous dirons qu'un automate est à états si les propriétés et les conditions portent sur les états. La figure 2.1 donne des variantes d'automates pour la formule $GFp \wedge GFq$. Les ensembles étiquetant les états et les transitions spécifient l'appartenance à un ensemble d'acceptation. Ici, nous avons deux ensembles d'acceptation. Nous pouvons déjà remarquer que les deux automates du haut sont les plus concis et montrer qu'il n'est pas possible d'obtenir un automate à un état en posant les conditions d'acceptation sur les états. Malgré les apparences, l'automate en haut à droite est de loin le meilleur : il est déterministe et n'a que 4 transitions. L'automate en haut et au milieu a en réalité 8 transitions. En effet, une transition étiquetée par une expression booléenne représente autant de transitions que l'expression a de solutions. Si on généralise la formule à $GFp_1 \wedge \cdots \wedge GFp_n$, les automates peuvent avoir de 1 à 2^n états, de 2^n à 2^{2n} transitions. Cet exemple illustre le fait que le choix de la variante d'automate joue un rôle important dans les méthodes de traduction. L'utilisation abusive d'expressions booléennes peut s'avérer un facteur agravant pour la vérification par la multiplication des transitions. Elle est cependant utile pour donner une représentation graphique ou textuelle simple et concise d'un automate. Afin de limiter l'explosion combinatiore du nombre d'états et de transitions dans le processus d'une vérification, une solution est de ne construire les éléments de l'automate qu'à la demande ; une autre solution est d'utiliser des représentations symboliques telles que les BDD.

Dans le cadre de la vérification probabiliste, nous serons amené à prendre en compte plusieurs qualités d'un automate : la non-ambiguïté et la séparation. La non-ambiguïté généralise la notion de déterministe : tout mot accepté par un automate non ambigu est la trace d'un seul chemin accepté. La séparation indique que les états d'un automate acceptent des langages disjoints. Dans la figure 2.1, les deux automates de droite sont non-ambigus et séparés, celui de gauche est non-ambigu et celui au milieu en haut est séparé. Remarquons qu'avec un seul ensemble d'acceptation, il n'est pas possible de produire un automate séparé pour la formule $GFp \wedge GFq$. Ceci montre encore une fois l'importance du choix du type d'automate pour traduire une formule.

Définition 2.2 *Soit $A = (S, S_0, T, \alpha, \beta, \lambda, Acc)$ un automate à transitions.*
- *A est non ambigu si tout mot infini accepté par A est la trace d'un chemin unique accepté par A.*
- *A est séparé si les états de l'automate acceptent des langages disjoints : $\forall s, s' \in S : s \neq s' \Rightarrow L(A, s) \cap L(A, s') = \emptyset$.*

2.2.2 Construction globale

Les premières traductions d'une formule LTL vers un automate reposent sur un même principe : un ensemble $el(f)$ caractérise les formules élémentaires

impliquées dans la vérification de la formule f, un état est un vecteur de booléens $s : el(f) \to \{0,1\}$ donnant des valeurs de vérité aux formules de $el(f)$ et les transitions sont définies pour respecter la sémantique de LTL. L'automate produit est étiqueté sur les états et a aussi des conditions d'acceptations sur les états. Nous proposons d'en donner une version couramment utilisée pour la vérification symbolique [9] et montrerons comment en modifiant légèrement la définition de $el(f)$, on obtient une construction produisant des automates à transitions.

Construction d'un automate à états. Dans les constructions classiques, l'ensemble des formules élémentaires $el(f)$ est composé des propositions élémentaires, des sous-formules de f du type Xg et des formules $X(gUh)$ où gUh est une sous-formule de f. Un état est un vecteur de booléens $s : el(f) \to \{0,1\}$ et la restriction de s à l'ensemble AP des propositions élémentaires définit une fonction d'étiquetage sur des états. Pour construire la relation de transitions, nous avons besoin de définir pour chaque état la fonction *sat* de satisfaction d'une formule. Intuitivement, $sat(s)(g)$ indique si la trace de tout chemin acceptant de source s satisfait la formule g. Cette relation est définie inductivement pour toute sous-formule de f par

- $sat(s)(g) = s(g)$ si $g \in el(f)$,
- $sat(s)(\neg g) = \neg sat(s)(g)$,
- $sat(s)(g \wedge h) = sat(s)(g) \wedge sat(s)(h)$,
- $sat(s)(gUh) = sat(s)(h) \vee (sat(s)(g) \wedge sat(s)(X(gUh)))$

Les transitions entre deux états s et s' sont posées pour respecter l'égalité $s(Xg) = sat(s')(g)$ pour toutes formules Xg de $el(f)$. Enfin les conditions d'acceptations sont définies pour s'assurer que les sous-formules du type gUh n'acceptent pas des séquences où h n'est jamais vérifié. Ainsi pour chaque sous-formule gUh, on définit un ensemble d'acceptation contenant les états s tels que $sat(s)(h) \vee \neg sat(s)(gUh) = 1$. Les états initiaux sont ceux vérifiant $s(f) = 1$.

Proposition 2.3 *Soit f une formule LTL sur AP. Soit $sub(f)$ l'ensemble des sous-formules de f. Posons $el(f) = AP \cup \{X(gUh) | gUh \in sub(f)\} \cup \{X(g) | X(g) \in sub(f)\}$. L'automate à états défini par :*

- $S = \{s : el(f) \to \{0,1\}\}$,
- $S_0 = \{s | sat(f)(s) = 1\}$,
- $T = \{s \to s' | \forall Xg \in el(f) : s(Xg) = sat(s')(g)\}$,
- $\forall s \in S, \forall p \in AP$, $\lambda(s)(p) = s(p)$,
- $Acc = \{Acc(gUh) | gUh \in sub(f)\}$ *avec* $Acc(gUh) = \{s \in S | sat(s)(h) \vee \neg sat(s)(gUh) = 1\}$,

accepte les mots infinis satifaisant la formule f.

Exemple 1 *Construisons l'automate pour la formule $f = pUq$. L'ensemble des formules élémentaires est $el(f) = \{p, q, X(pUq)\}$. Les états de l'automate sont donc des triplets de booléens. Dans une première étape, nous évaluons pour chaque état la valeur de la fonction de satisfaction pour la formule pUq. Ce calcul permet de déterminer complétement la relation de transition de l'automate : les états vérifiant pUq ont pour successeurs tous les états vérifiant*

s			sat(s)	acceptation
p	q	$X(pUq)$	pUq	$q \vee \neg(pUq)$
0	0	0	0	1
1	0	0	0	1
0	1	0	1	1
1	1	0	1	1
0	0	1	0	1
1	0	1	1	0
0	1	1	1	1
1	1	1	1	1

TAB. 2.1 – Calcul de l'automate (à états) pour la formule pUq

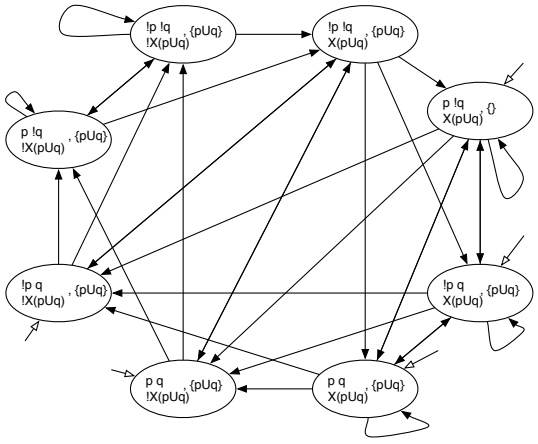


FIG. 2.2 – Automate (à états) pour la formule pUq

$X(pUq)$ et ceux ne vérifiant pas pUq ont pour successeurs les autres états. Dans une deuxième étape, nous calculons la valeur de la fonction de satisfaction pour la formule $q \vee \neg(pUq)$. Celle-ci nous donne les états appartenant à l'ensemble d'acceptation associé à la formule pUq. Finalement nous désignons les états vérifiant pUq comme états initiaux. Le tableau 2.1 donne les résultats des calculs nécessaires à la construction de l'automate et la figure 2.2 donne la représentation graphique de l'automate.

Construction d'un automate à transitions. Ma construction est une variante de cette construction classique. J'ai choisi comme formules élémentaires d'une formule f, la formule f, les sous-formules du type gUh et les formules g telles que Xg est une sous-formule de f. Notons $el_T(f)$ cet ensemble de fonctions élémentaires. La fonction de satisfaction *sat* d'une formule est définie pour toute étiquette $a \in 2^{AP}$ et tout état destination s'. Intuitivement, $sat_T(a, s')(g)$ indique si la trace d'un chemin acceptant commençant par une transition étiquetée par a et de destination s' vérifie la formule g. Cette relation est définie

inductivement par

- $sat_T(a, s')(Xg) = s'(g)$,
- $sat_T(a, s')(p) = a(p)$ si $p \in AP$,
- $sat_T(a, s')(\neg g) = \neg sat_T(a, s')(g)$,
- $sat_T(a, s')(g \wedge h) = sat_T(a, s')(g) \wedge sat_T(a, s')(h)$,
- $sat_T(a, s')(gUh) = sat_T(a, s')(h) \vee (sat_T(a, s')(g) \wedge sat_T(a, s')(X(gUh)))$

Une transition $s \xrightarrow{a} s'$ doit vérifier $s(g) = sat_T(a, s')(g)$ pour toute formule g de $el_T(g)$. Notez que la donnée d'un étiquetage et d'un état destination définit entièrement l'état source de la transition. Les conditions d'acceptation sont définies sur les transitions de manière analogue à la construction précédente. Pour chaque sous-formule gUh, on définit un ensemble d'acceptation contenant les transitions $s \xrightarrow{a} s'$ tel que $sat_T(a, s')(h) \wedge \neg s(gUh) = 1$. Les états initiaux sont ceux vérifiant $s(f)$.

Proposition 2.4 *Soit f une formule LTL sur AP. Soit $sub(f)$ l'ensemble des sous-formules de f. Posons $el_T(f) = \{f\} \cup \{gUh | gUh \in sub(f)\} \cup \{g | X(g) \in sub(f)\}$. L'automate à transitions défini par :*

- $S = \{s : el_T(f) \to \{0, 1\}\}$,
- $S_0 = \{s | s(f) = 1\}$,
- $T = \{s \xrightarrow{a} s' | \forall g \in el_T(f) : s(g) = sat_T(a, s')(g)\}$,
- $Acc = \{Acc(gUh) | gUh \in sub(f)\}$ *avec* $Acc(gUh) = \{s \xrightarrow{a} s' | sat_T(a, s')(h) \vee \neg s(gUh) = 1\}$,

accepte les mots infinis satifaisant la formule f.

Exemple 2 *Construisons l'automate à transitions pour la formule $f = pUq$. L'ensemble des formules élémentaires est $el_T(f) = \{pUq\}$. Le codage des états de l'automate se résume à un booléen. Dans une première étape, nous évaluons pour chaque valeur d'étiquette a et chaque état destination s', la valeur de la fonction de satisfaction pour la formule pUq. Ce calcul permet de déterminer complètement les états sources s des transitions à partir de l'étiquette a et de la destination s'. Dans une deuxième étape, nous calculons la valeur de la fonction de satisfaction pour la formule $q \vee \neg(pUq)$. Celle-ci nous donne les transitions appartenant à l'ensemble d'acceptation associé à la formule pUq. Finalement nous désignons les états vérifiant pUq comme états initiaux. Le tableau 2.2 donne les résultats des calculs nécessaires à la construction de l'automate et la figure 2.3 donne à gauche la représentation graphique de l'automate.*

Propriétés des automates. Les deux constructions sont extrêmement similaires et pourtant l'automate à transitions est toujours plus petit que l'automate à états. En effet remarquons que les ensembles de fonctions élémentaires des deux constructions servant à coder les états sont liés par l'identité $el(f) \cup \{f\} = X(el_T(f)) \cup AP$; ainsi le facteur de réduction est de $2^{|AP|}$ ou $2^{|AP|-1}$ suivant que f est dans $el(f)$. Par exemple, l'automate à transitions (figure 2.3) de la formule $rU(pUq)$ à 4 états et 32 transitions (12 transitions

a		s'	$s = sat_T(a, s')$	acceptation
p	q	pUq	pUq	$q \vee \neg(pUq)$
0	0	0	0	1
1	0	0	0	1
0	1	0	1	1
1	1	0	1	1
0	0	1	0	1
1	0	1	1	0
0	1	1	1	1
1	1	1	1	1

TAB. 2.2 – Calcul de l'automate à transitions pour la formule pUq

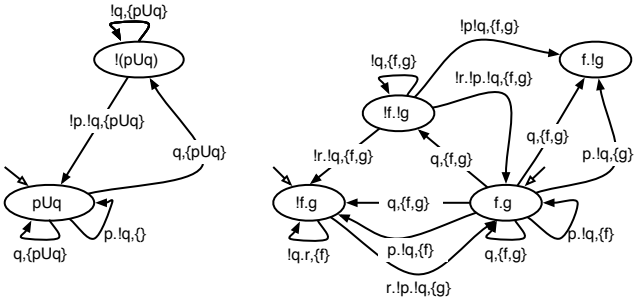


FIG. 2.3 – Automates à transitions pour les formules $f = pUq$ et $g = rU(pUq)$

symboliques) alors que l'automate à états correspondant a 32 états et 256 transitions. La proposition suivante donne l'ordre de grandeur des tailles des automates en fonction des caractéristiques de la formule.

Proposition 2.5 *Soit f une formule LTL sur AP. Notons $Temp(f)$ le nombre d'opérateurs X et U contenus dans la formule f. L'automate à états (produit par la construction classique) a au plus $2^{|AP|} \cdot 2^{Temp(f)}$ états et $2^{2\cdot|AP|} \cdot 2^{Temp(f)}$ transitions. L'automate à transitons a au plus $2^{Temp(f)+1}$ états et $2^{|AP|} \cdot 2^{Temp(f)+1}$.*

Malgré l'amélioration apportée à la construction classique, nous verrons dans la partie expérimentation (section 2.2.4) que cette construction est largement supplantée par les contructions locales. Cependant, l'automate produit possède des propriétés bien adaptées à la vérification probabiliste (voir la section 2.4). D'une part, l'automate est déterministe en arrière : un état n'est destination que d'une transition pour une valeur d'étiquette donnée. De plus, l'automate est non-ambigu. D'autre part, l'automate est séparé : deux états distincts de l'automate reconnaissent des langages disjoints. Ces propriétés sont aussi vérifiées par les automates produits par la construction classique.

Proposition 2.6 *Les deux constructions produisent des automates non-ambigus et séparés.*

Représentation symbolique. Les constructions globales se prêtent bien à des représentations symboliques de l'automate. En effet, les états et les transitions sont des vecteurs de booléens, et les différentes composantes de l'automate sont des ensembles de vecteurs de booléens représentables par des fonctions booléennes. De telles représentations sont couramment utilisées pour la vérification symbolique à base de BDD. Elles se caractérisent par leur efficacité à résoudre de nombreux problèmes de vérification de grande taille, mais aussi par leur simplicité de mise en œuvre. On pourra trouver dans [9] une version symbolique de la construction classique, ainsi que des algorithmes de vérification. Montrons que notre technique s'applique aussi à la construction d'un automate à transitions. Soit f une formule LTL. Considérons deux tableaux de variables booléennes *now* et *next* indexés par les formules de $el_T(f)$, et identifions les propositions élémentaires à des variables booléennes. Nous coderons la fonction caractéristique d'un ensemble d'états par une expression sur les variables *now* et celle d'un ensemble de transitions par une expression sur les trois jeux de variables. Interprétons toute sous-formule de g, comme une expressions booléenne $\Phi(g)$ sur les variables *now*, *next* et les propositions élémentaires :

- $\Phi(Xg) = next[g]$,
- $\Phi(p) = p$ si $p \in AP$,
- $\Phi(\neg g) = \neg\Phi(g)$,
- $\Phi(g \wedge h) = \Phi(g) \wedge \Phi(h)$.
- $\Phi(gUh) = now[gUh]$.

L'automate à transitions traduisant une formule LTL f est simplement défini par les expressions booléennes suivantes :

- L'ensemble des états initiaux : $S_0 = now[f]$,

– La relation de transition :

$$T = \bigwedge_{gUh \in el(f)} (now[gUh] \Leftrightarrow (\Phi(h) \vee (\Phi(g) \wedge next[gUh]))) \wedge \bigwedge_{g \in el(f)} (now[g] \Leftrightarrow \Phi(g))$$

– Les ensembles d'acceptation :

$$\forall gUh \in el_T(f) : Acc[gUh] = \neg now[gUh] \vee \Phi(h)$$

L'automate construit est rigousement identique à celui produit par la construction globale. Cependant sa formulation est bien plus simple pour construire et coder un automate à l'aide de BDD. Pour obtenir une représentation explicite de l'automate, il suffit de développer les fonctions booléennes. Cette opération n'est intéressante que dans un but pédagogique. En pratique, toutes les opérations utiles peuvent être réalisées sur la représentation symbolique, évitant ainsi une explosion combinatoire prématurée des algorithmes de vérification.

Exemple 3 *La représentaton symbolique de l'automate traduisant la formule* $g = rU(pUq)$ *est :*

– $S_0 = now[g]$,
– $T = (now[g] \Leftrightarrow (now[pUq] \vee (r \wedge next[g]))) \wedge (now[pUq] \Leftrightarrow (q \vee (p \wedge next[pUq])))$,
– $Acc[g] = \neg now[g] \vee now[f]$ *et* $Acc[pUq] = \neg now[pUq] \vee q$.

La figure 2.3 donne une représentation graphique de l'automate. Cependant cette représentation explicite n'est pas nécessaire pour réaliser des opérations sur celui-ci. A partir de l'expression des états initiaux, nous déduisons que l'automate a deux états initiaux : $[g, pUq]$, $[g, \neg(pUq)]$. *Les transitions successeurs d'un état pour une valeur des propositions sont obtenues par une opération de substitution sur la fonction* T. *Considérons l'état* $s = [g, pUq]$, *et l'étiquette donnée par* $p = 1, q = r = 0$. *En substituant les valeurs de l'état* s *(*$now[g] = now[pUq] = 1$*) et les valeurs des propositions, on obtient la liste des états successeurs de* s *sous la forme d'une expression booléenne sur les variables next. Le résultat est l'expression* $next[pUq]$ *et donc* s *a deux états successeurs :* $[g, pUq]$, $[\neg g, pUq]$. *L'appartenance des transitions aux ensembles d'acceptation est obtenue en évaluant les expressions booléennes* $Acc[g]$, $Acc[pUq]$. *Ainsi les deux transitions sont dans* $Acc[g]$ *et ne sont pas dans* $Acc[pUq]$.

Simplification de l'automate La construction globale peut être légèrement améliorée par trois techniques :

1. Ne garder que les états accessibles,
2. Eliminer les états ne reconnaissant aucun mot,
3. Réduire l'ensemble des états initiaux à un état.

Les deux premières techniques réduisent le nombre d'états et de transitions de l'automate et peuvent être réalisées directement sur sa représentation symbolique (voir [9]). Les états de l'automate seront alors symboliquement représentés par une expression booléenne sur les variables Now. Dans la dernière technique,

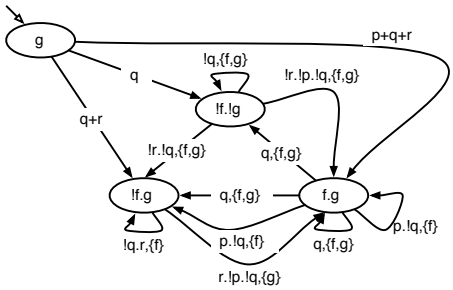


FIG. 2.4 – Automate à transitions simplifié pour la formule $rU(pUq)$

nous ajoutons à l'automate un état identifié par f, la formule à traduire. Cet état sera l'unique état initial, il n'aura pas de transition entrante et les transitions sortantes $f \xrightarrow{a} s'$ devront vérifier $sat_T(a, s')(f) = 1$; ces dernières pourront être représentées symboliquement par une expression sur les propositions élémentaires et les variables *next*. D'autre part, si f n'est pas de la forme gUh, elle n'est plus considérée comme une formule élémentaire. La figure 2.4 donne le résultat des simplifications sur l'automate représentant la formule $rU(pUq)$.

2.2.3 Construction locale

Le défaut des constructions globales est qu'elles produisent immédiatement un automate de taille exponentielle par rapport à la taille de la formule. L'exemple le plus frappant est la formule $X^n p$ (i.e. $X \dots Xp$ avec n opérateurs X). Cette formule indique que la nième action d'un comportement doit vérifier la proposition p. Intuitivement, il suffit d'un automate à transitions à $n+2$ états pour coder cette formule. Les constructions globales produisent pour cette formule des automates de taille exponentielle, même après simplification. En 1995, R. Gerth, D. Peled, M. Y. Vardi et P. Wolper proposent un nouveau type de la construction [37]. L'idée consiste à partir de la formule à vérifier, et à la développer sous la forme d'une disjonction de conjonction de formules ne faisant apparaître que des propositions élémentaires et des formules du type Xg. Les états initiaux sont donnés par les conjonctions du développement de la formule initiale ; les termes des conjonctions portant sur les propositions élémentaires définissent les propriétés à vérifier dans l'état, et ceux portant sur les formules du type Xg servent à construire les états successeurs. La construction continue en développant les formules associées aux formules représentant les états successeurs. Cette construction locale de l'automate d'une formule ne produit pas systématiquement les états correspondant à toutes les combinaisons possibles des formules élémentaires ; ceci rend cette construction intéressante en pratique. Un des points clefs de cette technique est comment capter les conditions d'acceptation relative aux formules du type gUh. Dans cette partie, nous allons rappeler les mécanismes de la construction locale de l'article fondateur [37], et montrer comment nous avons adapté et simplifié cette technique pour produire des automates à transitions.

Pour que les constructions locales soient applicables, les formules sont mises

sous une forme positive ; en d'autres termes, l'opérateur de négation ne peut être appliqué qu'aux propositions élémentaires. A cette fin, nous introduisons l'opérateur dual du until V : $gVh = \neg(\neg gU\neg h)$. Lors du développement d'une formule, nous sommes amenés à éliminer les formules du type gUh et gVh. Les éliminations sont réalisées grâce aux identités :

$$\begin{aligned} gUh &= h \vee (g \wedge X(gUh)) \\ gVh &= (g \wedge h) \vee (h \wedge X(gVh)) \end{aligned}$$

Bien que ces assertions semblent similaires, elles se distinguent par le fait que l'identité relative à la formule gUh ne tient pas compte de la propriété d'inévitabilité de h, alors qu'aucune propriété de ce type n'est nécessaire pour la validité de la formule gVh. Ainsi dans les deux constructions, nous associerons un ensemble d'acceptation à chaque sous-formule du type gUh. Les deux constructions locales se distingueront par le type d'automates produits, et aussi sur la manière de capter les conditions d'acceptation.

Construction d'un automate à états Nous allons donner une version simplifiée de la construction originale [37], captant autant que possible les optimisations proposées par les concepteurs. Le point clef de cette construction concerne la manière de développer une formule LTL. Ce développement ne produit pas une formule LTL, mais une expression symbolique représentant des ensembles d'ensembles de symboles. Les symboles utilisés sont :

$$\begin{aligned} symbole(f) &= AP \cup \{\neg p | p \in AP\} \\ &\cup \{now[gUh] | gUh \in sub(f)\} \\ &\cup \{now[h] | gUh \in sub(f)\} \\ &\cup \{next[gUh] | gUh \in sub(f)\} \\ &\cup \{next[gVh] | gVh \in sub(f)\} \\ &\cup \{next[g] | X(g) \in sub(f)\} \end{aligned}$$

Les états de l'automate produit sont des sous-ensembles de symboles. Les symboles $now[gUh]$ et $now[h]$ sont utilisés pour capter les conditions d'acceptation : un état ne contenant pas le symbole $now[gUh]$ ou contenant le symbole $now[h]$ est un état de l'ensemble d'acceptation associé à la formule gUh. A l'exception des formules gUh, les règles pour développer une formule sont celles attendues. Dans les expressions suivantes, $\Phi(g)$ est le résultat du développement d'une formule, les opérateurs $\cdot$ et $+$ sont des opérateurs d'union : $\cdot$ est l'union sur les ensembles de symboles et $+$ celui sur les ensembles d'ensembles de symboles :

- $\Phi(p) = p$ si $p \in AP$,
- $\Phi(\neg p) = \neg p$ si $p \in AP$,
- $\Phi(g \wedge h) = \Phi(g) \cdot \Phi(h)$,
- $\Phi(g \vee h) = \Phi(g) + \Phi(h)$,
- $\Phi(Xg) = next[g]$,
- $\Phi(gUh) = now[h] \cdot \Phi(h) + now[gUh] \cdot \Phi(g) \cdot next[gUh]$,

– $\Phi(g\,V\,h) = \Phi(g) \cdot \Phi(h) + \Phi(h) \cdot next[g\,V\,h]$.

A partir de la définition de l'ensemble des symboles et la fonction de développement d'une formule, la définition de l'automate d'une formule proposée dans [37] est résumée par la proposition suivante :

Proposition 2.7 *Soit f une formule LTL sur AP. L'automate à états défini par :*

– $S = \{s \subseteq symbole(f)\}$,
– $S_0 = \Phi(f)$,
– $T = \{s \rightarrow s' | s' \in \prod_{next[h] \in s} \Phi(h)\}$,
– $\forall s \in S,\ \lambda(s) = \bigwedge_{p \in AP \cap s} p \wedge \bigwedge_{p \in AP : \neg p \in s} \neg p$,
– $Acc = \{Acc(g\,U\,h) | g\,U\,h \in sub(f)\}$ *avec* $Acc(g\,U\,h) = \{s \in S | now[h] \in s \vee now(g\,U\,h) \notin s\}$,

accepte les mots infinis satisfaisant la formule f.

Exemple 4 *Calculons l'automate à états pour la formule $g = r\,U\,(p\,U\,q)$. Notons que la formule est déjà sous une forme positive. Posons $f = p\,U\,q$. L'automate a deux ensembles d'acceptation $Acc = \{f, g\}$. Développons la formule g :*

$$\begin{aligned} \Phi(g) &= now[f] \cdot \Phi(f) + now[g] \cdot r \cdot next[g] \\ &= now[f] \cdot now[q] \cdot q + now[f] \cdot p \cdot next[f] + now[g] \cdot r \cdot next[g] \end{aligned}$$

L'automate à états possède donc trois états initiaux :

$$S_0 = \{\{now[f], now[q], q\}, \{now[f], p, next[f]\}, \{now[g], r, next[g]\}\}$$

Notons que le premier état n'a pas de symboles next ; cet état a $\emptyset$ comme état successeur. Le calcul des successeurs du deuxième état nécessite le développement de la formule f :

$$\Phi(f) = now[q] \cdot q + now[f] \cdot p \cdot next[f]$$

Remarquons que ce calcul produit un nouvel état $\{now[q]\}$ dont l'unique successeur est $\emptyset$. D'autre part, le troisième état initial a pour successeur les trois états initiaux. La figure 2.5 donne une représentation graphique de l'automate résultant. Les concepteurs de cette méthode ont proposé de nombreuses heuristiques pour simplifier l'automate, Par exemple, il est possible de fusionner les états $\{now[f], now[q], q\}$ et $\{now[q], q\}$ en appliquant la règle suivante : si $now[g\,U\,h]$ et $now[h]$ appartiennent à un état s, retirer de s, $now[g\,U\,h]$.

Construction d'un automate à transitions La construction que j'ai proposée dans [14] repose sur deux simplifications de la construction précédente : (1) le développement d'une formule LTL produit un ensemble de transitions ; (2) nous avons écarté le symbole $now[h]$ dans le développement des formules $g\,U\,h$. La simplification (1) est naturelle quand l'intention est de produire des automates à transitions. La simplification (2) traduit le fait que la seule information

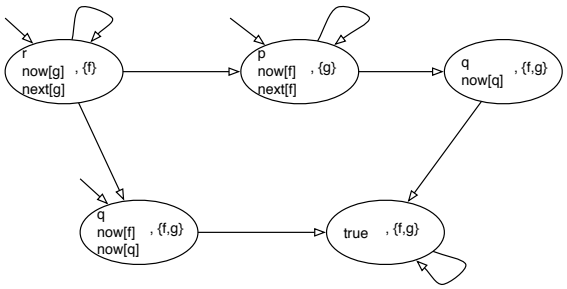


FIG. 2.5 – Automate à états pour la formule $g = r\,U\,(p\,U\,q)$

utile dans le développement d'une formule $g\,U\,h$ est : est-ce que la propriété d'inévitabilité restera à vérifier ? Plutôt qu'utiliser le symbole $now[g\,U\,h]$, nous préférons le symbole $nacc[g\,U\,h]$ indiquant clairement la non acceptation d'une transition. L'ensemble des symboles employés pour le développement d'une formule est donc :

$$\begin{aligned} symbole_T(f) &= AP \cup \{\neg p | p \in AP\} \\ &\cup \{nacc[g\,U\,h] | g\,U\,h \in sub(f)\} \\ &\cup \{next[g\,U\,h] | g\,U\,h \in sub(f)\} \\ &\cup \{next[g\,V\,h] | g\,V\,h \in sub(f)\} \\ &\cup \{next[g] | X(g) \in sub(f)\} \end{aligned}$$

Les états de l'automate à transitions sont des ensembles de sous-formules de la formule. Le calcul des transitions successeurs d'un état est obtenu en développant les sous-formules représentant l'état. Dans les expressions suivantes, notez la modification de la règle relative aux formules $g\,U\,h$:

- $\Phi_T(p) = p$ si $p \in AP$,
- $\Phi_T(\neg p) = \neg p$ si $p \in AP$,
- $\Phi_T(g \wedge h) = \Phi_T(g) \cdot \Phi_T(h)$,
- $\Phi_T(g \vee h) = \Phi_T(g) + \Phi_T(h)$,
- $\Phi_T(X g) = next[g]$,
- $\Phi_T(g\,U\,h) = \Phi_T(h) + nacc[g\,U\,h] \cdot \Phi_T(g) \cdot next[g\,U\,h]$,
- $\Phi_T(g\,V\,h) = \Phi_T(g) \cdot \Phi_T(h) + \Phi_T(h) \cdot next[g\,V\,h]$.

Un point essentiel de la méthode est que les symboles peuvent être manipulés comme des variables booléennes, les opérateurs $+$ et $\cdot$ comme les opérateurs booléens $\vee$ et $\wedge$. Notez que l'expression booléenne représentant le développement d'une formule peut toujours s'écrire sous la forme disjonctive-conjonctive où les variables du type *next* et *nacc* apparaissent positivement. L'intérêt de ce point de vue est que ces expressions acceptent les simplifications classiques des expressions booléennes, et ainsi il est possible de réduire naturellement le nombre de transitions issues d'un état. A partir de la définition du développement d'une formule, la définition donnant l'automate à transitions est donnée par la proposition suivante :

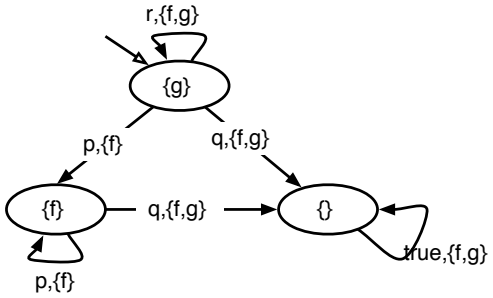


FIG. 2.6 – Automate à transitions pour la formule $g = r\,U\,(p\,U\,q)$

Proposition 2.8 *Soit f une formule LTL sur AP. Soit $sub(f)$ l'ensemble des sous-formules de f. Posons $el_T(f) = \{f\} \cup \{gUh | gUh \in sub(f)\} \cup \{g | X(g) \in sub(f)\}$ et $Acc(f) = \{Acc(gUh) | gUh \in sub(f)\}$. Pour tout sous-ensemble de formules s de $el_T(f)$, considérons la décomposition suivante de $\prod_{g\in s} \Phi_T(g)$:*

$$\prod_{g\in s} \Phi_T(g) = \sum_{(K,Nacc,Next)\in Terme_s} \left(K \cdot \prod_{h\in Nacc} nacc[h] \cdot \prod_{h\in Next} next[h] \right)$$

où $Terme_s \subseteq 2^{2^{AP}} \times \{gUh | gUh \in sub(f)\} \times el_T(f)$. L'automate à transitions défini par :

- *$S = \{s \subseteq el_T(f)\}$,*
- *$S_0 = \{f\}$,*
- *$T = \{s \xrightarrow{K,acc} s' | (K, Acc(f) \setminus acc, s') \in Terme_s\}$,*
- *$\lambda : T \to 2^{2^{AP}}$ est donné par $\lambda(s \xrightarrow{K,acc} s') = K$,*
- *$Acc = Acc(f)$ avec $Acc(gUh) = \{s \xrightarrow{K,acc} s' \in T | gUh \in acc\}$,*

accepte les mots infinis satisfaisant la formule f.

Exemple 5 *Calculons l'automate à transitions pour la formule $g = r\,U\,(p\,U\,q)$. Posons $f = p\,U\,q$. L'automate a un état initial $\{g\}$. Les transitions successeurs sont obtenues en développant la formule g :*

$$\begin{aligned} \Phi_T(g) &= \Phi(f) + nacc[g] \cdot r \cdot next[g] \\ &= q + nacc[f] \cdot p \cdot next[f] + nacc[g] \cdot r \cdot next[g] \end{aligned}$$

L'état initial est la source de trois transitions successeurs : $\{g\} \xrightarrow{q,\{f,g\}} \emptyset$, $\{g\} \xrightarrow{p,\{g\}} \{f\}$, $\{g\} \xrightarrow{r,\{f\}} \{g\}$. La construction de l'automate continue avec le développement de la formule f :

$$\Phi_T(f) = q + nacc[f] \cdot p \cdot next[f]$$

Ce calcul produit deux nouvelles transitions mais pas de nouveau état. La figure 2.6 donne la représentation graphique de l'automate résultant.

Simplification de l'automate Nous pouvons appliquer 3 types de simplifications, chacune compatible avec la construction à la demande :

1. Quand deux états ont les mêmes expressions $\prod_{g \in s} \Phi_T(g)$ représentant la liste de leurs transitions successeurs, ces deux états peuvent être fusionnés. Cette simplification est simple à implémenter quand on utilise des BDD. En effet, il suffit d'identifier tout état par l'expression booléenne de ses transitions successeurs.
2. Il existe de nombreuses manières d'écrire une expression sous la forme disjonctive conjonctive. Celle qui donne le meilleur résultat consiste à spécialiser le calcul des transitions successeurs pour une valeur donnée des propositions élémentaires. Il suffit de substituer ces valeurs dans l'expression et d'en calculer les impliquants premiers. Notez que cette technique est bien adaptée pour les méthodes de vérification à la volée. En effet, pour construire les transitions successeurs d'un état (s, q) du produit synchronisé d'un système et de l'automate d'une formule, nous devons construire pour toute transition $s \xrightarrow{a} s'$ du système, tous les successeurs de l'état q pour la valeur a des propositions élémentaires.
3. Avant de lancer la construction de l'automate, il est parfois utile de simplifier la formule afin de réduire autant que possible le nombre de sous formules élémentaires.

La table 2.3 donne quelques résultats expérimentaux sur l'influence des optimisations. Nous avons appliqué la construction locale pour toutes les combinaisons possibles d'optimisation, sur une centaine de formules tirées aléatoirement dont la taille varie de 15 à 20. Les colonnes *Automate* donnent la taille cumulée des automates ; les colonnes *Produit synchronisé* donnent la taille cumulée des produit synchronisés des automates des formules avec un système comprenant 200 états. Nous noterons que les simplifications (1) et (3) réduisent la taille de l'automate, alors que la simplification (2) a pour objectif de rendre l'automate plus déterministe et ainsi de réduire le nombre de transitions du produit synchronisé. Les mêmes expérimentations appliquées à 39 formules "classiques" de LTL (voir la table 2.3) aboutissent aux mêmes conclusions.

2.2.4 Expérimentations

La conception de nouvelles constructions d'automates traduisant une formule LTL reste encore un sujet de recherche très actif [24, 76, 73, 35, 38, 80, 74, 78, 67, 34]. Il est naturel de se demander si la construction que j'ai proposée en 1999, est encore d'actualité. Nous ne pouvons donner à cette question qu'une réponse partielle. En effet, la comparaison dépend en grande partie de la qualité des outils qui les implémentent. Par exemple, l'outil prototype Modella illustrant les travaux de [74] n'est pas suffisamment fiable pour être pris en compte. Enfin, d'autres outils ne sont plus disponibles. Nous avons donc réduit notre étude comparative aux outils suivants :

– Spot [25] est une bibliothèque C++ intégrant mes deux constructions d'automates, avec leurs optimisations. Nous désignerons par Spot (LA-

Optimisation	Automate		Produit synchronisé	
	Etats	Transitions	Etats	Transitions
rien	1162	4458	231971	11344438
(1)	1023	3611	204247	9601659
(2)	1162	4458	229575	8509410
(3)	1056	4000	211001	10170928
(1)+(2)	1023	3611	202877	7544852
(1)+(3)	961	3331	192050	8790687
(2)+(3)	1056	4000	208817	7563089
(1)+(2)+(3)	961	3331	190684	6831836

TAB. 2.3 – Influence des optimisations sur 100 formules tirées aléatoirement

Optimisation	Automate		Produit synchronisé	
	Etats	Transitions	Etats	Transitions
rien	250	1735	46387	3360454
(1)	189	622	37200	2271545
(2)	250	1735	46110	1951413
(3)	217	1599	39788	2816419
(1)+(2)	189	622	37174	1580669
(1)+(3)	165	527	32400	1865274
(2)+(3)	217	1599	39511	1590360
(1)+(2)+(3)	165	527	32374	1310746

TAB. 2.4 – Influence des optimisations sur 39 formules classiques

CIM), l'outil utilisant la construction globale et par Spot (FM), celui basé sur la construction locale.
- Spin [48] est un des outils de référence pour la vérification de système. Il intègre un module de construction d'automates basé sur la première construction locale [37], enrichi par des techniques de simplification d'automates. L'automate produit est un automate à états n'ayant qu'un seul ensemble d'acceptation.
- LBT est un programme de construction d'automates intégré dans un outil de vérification, l'outil Maria [65]. Il est basé sur la construction locale [37].
- Wring est un prototype illustrant les travaux de F. Sommenzi et R. Bloem. Il est basé sur [37], intègre un module de simplification de la formule à traduire et un module de réduction de l'automate construit.
- LTL2BA est un outil illustrant les travaux de P. Gastin et D. Oddoux [35, 67]. Cette construction passe par des automates alternants faibles pour produire l'automate d'une formule ; elle intègre aussi un module de simplification de la formule à traduire et un module de réduction de l'automate produit. L'outil génère des automates à transitions.

La table 2.5 donne les résultats expérimentaux de notre étude comparative des outils. Celle-ci a été réalisée grâce à l'outil de test LBTT [79]. Nous avons lancé les constructions sur 100 formules générées aléatoirement dont la taille varie de 15 à 20. Les colonnes *Automate* donnent la taille cumulée des automates ; les colonnes *Produit synchronisé* donnent la taille cumulée des produits synchronisés des automates des formules avec un système comprenant 200 états. Notons que la construction globale (Spot (LACIM)) n'est pas très efficace. Pour devenir intéressante, la construction locale (LBT) doit intégrer des modules de simplification de formules et d'automates (SPIN, Wring). Les meilleures constructions sont celles produisant des automates à transitions (Spot (FM), LTL2BA). Cependant Spot (FM) se distingue de LTL2BA par le nombre de transitions du produit synchronisé. Ces expérimentations prouvent que malgré sa simplicité, la construction locale que j'ai proposée en 1999, reste efficace en comparaison avec les autres techniques. Une question intéressante serait d'évaluer l'influence des techniques de réduction d'automates de LTL2BA dans Spot (FM). Notons que ces réductions se feraient aux détriments du caractère à la demande de la construction locale.

La table 2.6 donne les résultats expérimentaux lorsque les automates produits par les outils sont transformés en automates compatibles avec l'outil SPIN. Il est surprenant de constater que les outils LTL2BA et Spot (FM) donnent encore les meilleurs résultats. Nous aboutissons aux mêmes conclusions lorsque l'expérimentation est appliqué à des formules classiques (voir les tables 2.5 et 2.8). Notez que nous avons intégré l'outil Modella dans ces nouvelles expérimentations.

2.3 Tester le vide d'un ω-automate

Dans les méthodes de vérification à base d'automates, on construit le produit synchronisé du système M et de l'automate traduisant la négation de la formule

Outils	Automate		Produit synchronisé	
	Etats	Transitions	Etats	Transitions
Spot (LACIM)	2675	40207	535000	32386360
Spot (FM)	961	3331	190684	6831836
Spin	2044	13924	398617	27056967
LBT	6502	38241	1281565	81134772
Wring	1793	4917	356716	16767017
LTL2BA	1004	3591	200729	10310947

TAB. 2.5 – Comparaison des outils de constructions sur 100 formules tirées aléatoirement

Outils	Automate		Produit synchronisé	
	Etats	Transitions	Etats	Transitions
Spot (LACIM)	5016	80135	1002558	53598468
Spot (FM)	1196	4601	235994	8642998
Spin	2044	13924	398617	27056967
Wring	1967	5565	391470	18690241
LTL2BA	1175	4577	234870	12759396

TAB. 2.6 – Comparaison des outils de constructions pour Spin sur 100 formules tirées aléatoirement

Outils	Automate		Produit synchronisé	
	Etats	Transitions	Etats	Transitions
Spot (LACIM)	428	3872	84800	4580084
Spot (FM)	165	527	32374	1310746
Spin	277	1196	54200	3548935
LBT	844	5584	161040	12530060
Wring	276	1480	50986	3041195
LTL2BA	169	506	33200	1941344
Modella	283	867	55689	2119971

TAB. 2.7 – Comparaison des outils de constructions d'automates sur 39 formules classiques

Outils	Automate		Produit synchronisé	
	Etats	Transitions	Etats	Transitions
Spot (LACIM)	993	11182	174791	8198966
Spot (FM)	222	904	43119	1813690
Spin	277	1196	54200	3548935
Wring	407	5339	52386	3112719
LTL2BA	205	624	40200	2480261
Modella	283	867	55689	2119971

TAB. 2.8 – Comparaison des outils de constructions d'automates pour Spin sur 39 formules classiques

à vérifier. Ce produit synchronisé est essentiellement le produit cartésien des deux automates dans lequel on ne garde que les paires de transitions étiquetées par le même label ; d'un point de vue automate, il reconnaît précisément les comportements (ou les traces des comportements) du système ne vérifiant pas la formule f.

Définition 2.9 *Soit* $M = \langle S, S_0, T_M, \alpha_M, \beta_M, \lambda_M \rangle$ *un système. Soit* $A = \langle Q, Q_0, T_A, \alpha_A, \beta_A, \lambda_A, Acc \rangle$ *un automate à transitions. Notons,* la projection sur M *(resp.* la projection sur A*) la fonction* π_M *(resp.* π_A*) définie sur* $(S \times Q) \cup (T_M \times T_A)$ *par* $\pi_M(z_M, z_A) = z_M$ *(resp.* $\pi_A(z_M, z_A) = z_A$*). Le* produit synchronisé *de* M *et* A *est l'automate à transitions* $M \otimes A = \langle S \times Q, S_0 \times Q_0, T_\otimes, \alpha_\otimes, \beta_\otimes, \lambda_\otimes, Acc_\otimes \rangle$ *où :*

- $T_\otimes = \{(t_M, t_A) \in T_M \times T_A \mid \lambda_M(t_M) = \lambda_A(t_A)\}$*, et*
- $\alpha_\otimes$ *et* $\beta_\otimes$ *sont définies par* $\alpha_\otimes(t_M, t_A) = (\alpha_M(t_M), \alpha_A(t_A))$ *et* $\beta_\otimes(t_M, t_A) = (\beta_M(t_M), \beta_A(t_A))$*, et*
- $\lambda_\otimes$ *est la restriction de* π_M *à* $T_\otimes$*, et*
- $Acc_\otimes$ *est égal à* $\pi_A^{-1}(Acc)$*.*

Notez que dans notre définition, le produit synchronisé est un automate à transitions dont les étiquettes sont des transitions du système ; ainsi cet automate reconnaît des mots infinis sur l'alphabet T_M. Le produit synchronisé, le système et l'automate sont liés par l'égalité :

$$L(M \otimes A) = Path^\omega(M) \cap \lambda_M^{-1}(L(A)).$$

où $L(M \otimes A)$ est l'ensemble des mots infinis de transitions de M reconnus par le produit, $Path^\omega(M)$ est l'ensemble des comportements infinis de M et $\lambda_M^{-1}(L(A))$ est l'ensemble des mots infinis de transitions dont la trace est acceptée par A. Ainsi le problème de la vérification revient à déterminer si le langage $L(M \otimes A)$ est vide. Ce problème se ramène à rechercher dans l'automate $M \otimes A$ un cycle contenant au moins une transition de chaque ensemble d'acceptation et accessible à partir d'un état initial. Notons qu'il n'est pas essentiel de considérer tous les cycles de l'automate ; il est suffisant de rechercher une composante fortement connexe, accessible à partir d'un état initial et contenant au moins une transition de chaque ensemble d'acceptation.

La recherche de composantes fortement connexes est traditionnellement réalisée en appliquant l'algorithme de Tarjan [77, 81]. Cet algorithme est basé sur un parcours en profondeur. Il utilise deux étiquettes associées aux sommets parcourus, NFNUMBER et LOWLINK : NFNUMBER donne un numéro d'ordre de la première visite du sommet et LOWLINK est utilisée pour caractériser des représentants remarquables des composantes fortement connexes. Pendant le parcours du graphe, les valeurs des NFNUMBER et LOWLINK sont mises à jour. Quand un sommet vérifiant NFNUMBER=LOWLINK est dépilé, la composante fortement connexe associée au sommet est traitée et les sommets retirés du graphe par un second parcours en se limitant aux sommets non effacés. Le défaut de cet algorithme est qu'une composante fortement connexe doit être complètement parcourue avant d'être traitée. En particulier, l'algorithme ne peut détecter un cycle acceptant à la volée, c'est-à-dire arrêter le calcul dès que le graphe parcouru contient un comportement acceptant.

Le nouvel algorithme présenté dans cette section est une simple variation de l'algorithme de Tarjan. La variable LOWLINK est remplacée par une pile de numéros de visite des racines de composantes fortement connexes du graphe parcouru. Nous stockons en plus dans cette pile les ensembles de conditions d'acceptation contenues dans chaque composante. Pendant le parcours en profondeur, la pile est mise à jour et à tout moment les composantes et leurs ensembles de conditions sont connus. L'algorithme détecte un comportement acceptant à la volée, et le parcours est arrêté dès qu'une des composantes contient au moins une transition pour chaque condition d'acceptation.

Une description de l'algorithme est donnée figure 2.7. L'automate à transitions est codé par :

- $S0$ est l'état initial,
- ACC est l'ensemble des conditions d'acceptation,
- $RELATION \subseteq S \times ACC \times S$ définit la relation de l'automate.

Notez que nous avons limité notre étude à un état initial. Un algorithme traitant un ensemble d'états initiaux peut être facilement déduit de notre algorithme. D'autre part, les ensembles d'acceptation sont définis par un étiquetage des arcs. Nous n'avons pas représenté les valeurs des propositions élémentaires, car celles-ci ne sont pas significatives pour la détection d'un cycle acceptant.

L'algorithme utilise les données suivantes pour la gestion du parcours et pour la mémorisation de la pile des racines des composantes fortement connexes du graphe parcouru :

- Num donne le nombre de sommet du graphe parcouru ; il est utilisé pour initialiser le numéro de visite d'un sommet lors de sa première visite,
- $Hash$ est une table (de hachage) contenant des couples $(node, integer)$; elle stocke les sommets parcourus avec le numéro de visite. Un numéro de visite égal à zéro signale un sommet parcouru, effacé du graphe.
- $Root$ est une pile de couples $(integer, accepting\,condition\,set)$; elle stocke les racines des composantes ainsi que les conditions d'acceptations contenues dans leurs composantes.
- Arc est une pile d'ensemble de conditions ; elle stocke les conditions des arcs reliant les composantes représentées dans la pile $Root$.

Pendant l'exécution de l'algorithme, nous pouvons considérer :

```
1 Check(){
2    Num = 1 ;
3    Hash.put(S0,1)
4    Root.push(1,EMPTYSET) ;
5    Arc.push(EMPTYSET) ;
6    Explore(S0,1) ;
7 }

8 Explore(Node v,int vorder){
9    for all (A,w) such that (v,A,w) in RELATION do {
10       (b,worder) = Hash.get(w) ;
11       if not(b) then {
12       (* w is a new explored node *)
13         Num = Num+1 ;
14          Hash.put(w,Num) ;
15          Root.push(Num,EMPTYSET) ;
16          Arc.push(A) ;
17          Explore(w,Num) ;
18       }
19       else if (worder <> 0) then {
20       (* w is a node of the current graph *)
21          (i,B) = Root.pop() ; B = B union A ;
22           while i>worder do {
23              A = Arc.pop() ; B = B union A ;
24              (i,A) = Root.pop() ; B = B union A ;
25          }
26          Root.push(i,B) ;
27          If B == ACC then report accepted cycle ;
28       }
29    }
30    (i,B) = Root.top() ;
31    if vorder == i then {
32    (* v is the root of a strongly connected component *)
33       Num = i-1 ;
34       Root.pop() ;
35       Arc.pop() ;
36       Remove(v) ;
37    }
38 }

39 Remove(Node v){
40    b=Hash.testset0(v) ;
41    if b then
42       for all (A,w) such that (v,A,w) in RELATION do
43          Remove(w) ;
44 }
```

FIG. 2.7 – Algorithme de vérification

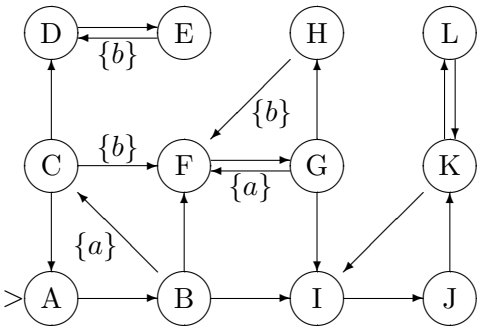


FIG. 2.8 – Un automate à transitions avec $Acc = \{a, b\}$

- le graphe effacé : le sous graphe contenant les sommets stockés dans $Hash$ de numéro de visite nul,
- le graphe courant : le sous graphe des sommets parcourus dont le numéro de visite est non nul,
- le chemin courant : le chemin stocké dans la pile induit par le parcours récursif en profondeur du graphe.

L'algorithme a été conçu pour préserver les propriétés suivantes à chaque fois qu'un nouvel arc est parcouru :

Propriété 1. Le graphe effacé ne contient pas de cycle acceptant ;

Propriété 2. La pile *Root* ne contient que des sommets du chemin courant, rangés dans le même ordre ;

Propriété 3. Si $Root = (o_1, A_1)(o_2, A_2)\dots(o_p, A_p)$ alors $o_1 = 1$ et la composante fortement connexe C_i du graphe courant est donnée par

$$\forall i < p : C_i = \{v \in \textit{graphe courant} / o_i \leq \textit{ordre}(v) \leq o_{i+1} - 1\}$$

et

$$C_p = \{v \in \textit{graphe courant} / o_p \leq \textit{ordre}(v) \leq Num\}$$

Propriété 4. Si la pile *Arc* contient $B_1 \cdot B_2 \cdots B_p$, les composantes fortement connexes sont reliées dans le graphe courant par des arcs $(order^{-1}(o_1 - 1), B_i, order^{-1}(o_i))$ avec $i > 1$. B_1 est un ensemble de conditions représentant un arc artificiel connecté à l'état initial, dont la valeur est toujours $\emptyset$.

Exemple 6 *Pour illustrer les caractéristiques de l'algorithme, appliquons le à l'automate de la figure 2.8. L'automate contient deux conditions ($ACC = \{a, b\}$). La table 6 donne les valeurs des données pour chaque étape de l'algorithme. Notez que les propriétés 1-4 attendues par l'algorithme sont vérifiées.*

1. *L'initialisation, lignes 2-6, est exécutée et la fonction $Explore(A, 1)$ est appelée.*

2. *Parcourons l'arc* $A \rightarrow B$. B *est un nouveau sommet,* ($Hash.get(B)$ *retourne* $(false, undefined)$*). Exécutons les lignes 12-17. Le sommet* B *définit une nouvelle composante fortement connexe du graphe parcouru : son numéro de visite est empilé dans Root et l'ensemble de conditions de l'arc* $A \rightarrow B$ *est empilé dans Arc.*

3. *Parcourons l'arc* $B \rightarrow C$. C *est aussi un nouveau sommet et les lignes 12-17 sont exécutées.*

4. *Parcourons* $C \rightarrow A$. A *a pour numéro de visite* 1 *(*$Hash.get(A)$ *retourne* $(true, 1)$*). Exécutons les lignes 20-27. Root et Arc sont dépilés jusqu'à ce que la tête de la pile Root contienne un numéro de visite plus petit ou égal à* 1*. Cette opération provoque la fusion des composantes* $\{A\}$, $\{B\}$ *et* $\{C\}$*. L'ensemble des conditions de la composante* $\{A, B, C\}$ *est mise à jour à* $\{a\}$*. Ligne 27,* $\{a\} \neq \{a, b\}$ *et aucun cycle acceptant n'est détecté.*

5. *Parcourons* $C \rightarrow D$. D *est un nouveau sommet.*

6. *Parcourons* $D \rightarrow E$. E *est encore un nouveau sommet.*

7. *Parcourons* $E \rightarrow D$. D *est un sommet de numéro de visite* 4*. Exécutons les lignes 20-27. Root et Arc sont dépilés jusqu'à ce que la tête de la pile Root contienne un numéro de visite plus petit ou égal à* 4*. Cette opération provoque la fusion des composantes* $\{D\}$ *et* $\{E\}$*.*

8. *A cette étape, tous les successeurs du sommet* E *ont été traversés. Lignes 30-31, le programme teste si* E *est une racine de composante fortement connexe. Ce n'est pas le cas et donc le sommet* E *est ignoré.*

9. *A cette étape, tous les successeurs du sommet* D *ont été traversés. Lignes 30-31,* E *est vu comme une racine de composante fortement connexe. L'exécution des lignes 32-36 provoque l'élimination des sommets de la composante* $\{D, E\}$ *du graphe courant.*

10. *Parcourons* $C \rightarrow F$. F *est un nouveau sommet.*

11. *Parcourons* $F \rightarrow G$. G *est un nouveau sommet.*

12. *Parcourons* $G \rightarrow F$. F *est un sommet de numéro de visite* 4*. Exécutons les lignes 20-27. Elles provoquent la fusion des composantes* $\{F\}$ *et* $\{G\}$*. A cette étape, l'ensemble des conditions de la composante est* $\{a\}$*.*

13. *Parcourons* $G \rightarrow H$. H *est un nouveau sommet.*

14. *Parcourons* $H \rightarrow F$. F *est un sommet de numéro de visite* 4 *. Exécutons les lignes 20-27. Elles provoquent la fusion des composantes* $\{F, G\}$ *et* $\{H\}$*. Le calcul de l'ensemble des conditions donne* $\{a, b\}$*. Ligne 27, un cycle acceptant est détecté et l'algorithme s'arrête :* $\{F, G, H\}$ *contient un cycle acceptant.*

Proposition 2.10 *Si un automate à transitions contient un cycle acceptant accessible à partir de l'état initial, l'algorithme de vérification détecte une composante fortement connexe contenant un cycle acceptant. De plus, l'algorithme s'arrête dès que le graphe parcouru contient un cycle acceptant.*

	Nœeud	Transition	Root	Arc	Hash
1	A.1		1.$\emptyset$	$\emptyset$	A.1
2	A.1	A→B	1.$\emptyset$, 2.$\emptyset$	$\emptyset$ $\emptyset$	A.1, B.2
3	B.2	B→C	1.$\emptyset$, 2.$\emptyset$, 3.$\emptyset$	$\emptyset$ $\emptyset$ {a}	A.1, B.2, C.3
4	C.3	C→A	1.{a}	$\emptyset$	A.1, B.2, C.3
5	C.3	C→D	1.{a}, 4. $\emptyset$	$\emptyset$ $\emptyset$	A.1, B.2, C.3, D.4
6	D.4	D→E	1.{a}, 4.$\emptyset$, 5.$\emptyset$	$\emptyset$ $\emptyset$ $\emptyset$	A.1, B.2, C.3, D.4, E.5
7	E.5	E→D	1.{a}, 4.{b}	$\emptyset$ $\emptyset$	A.1, B.2, C.3, D.4, E.5
8	E.5		1.{a}, 4.{b}	$\emptyset$ $\emptyset$	A.1, B.2, C.3, D.4, E.5
9	D.4		1.{a}	$\emptyset$	A.1, B.2, C.3, D.0, E.0
10	C.3	C→F	1.{a}, 4.$\emptyset$	$\emptyset$ {b}	A.1, B.2, C.3, F.4, D.0, E.0
11	F.4	F→G	1.{a}, 4.$\emptyset$,5.$\emptyset$	$\emptyset$ {b} $\emptyset$	A.1, B.2, C.3, F.4, G.5, D.0, E.0
12	G.5	G→F	1.{a}, 4.{a}	$\emptyset$ {b}	A.1, B.2, C.3, F.4, G.5, D.0, E.0
13	G.5	G→H	1.{a}, 4.{a}, 6.$\emptyset$	$\emptyset$ {b} $\emptyset$	A.1, B.2, C.3, F.4, G.5, H.6, D.0, E.0
14	H.6	H→G	1.{a}, 4.{a,b}	$\emptyset$ {b}	A.1, B.2, C.3, F.4, G.5, H.6, D.0, E.0

TAB. 2.9 – Une exécution de l'algorithme de vérification

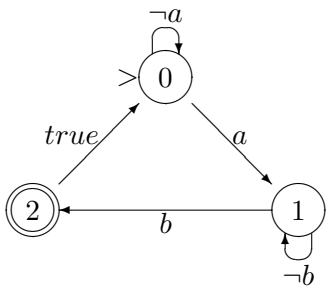


FIG. 2.9 – Automate compteur

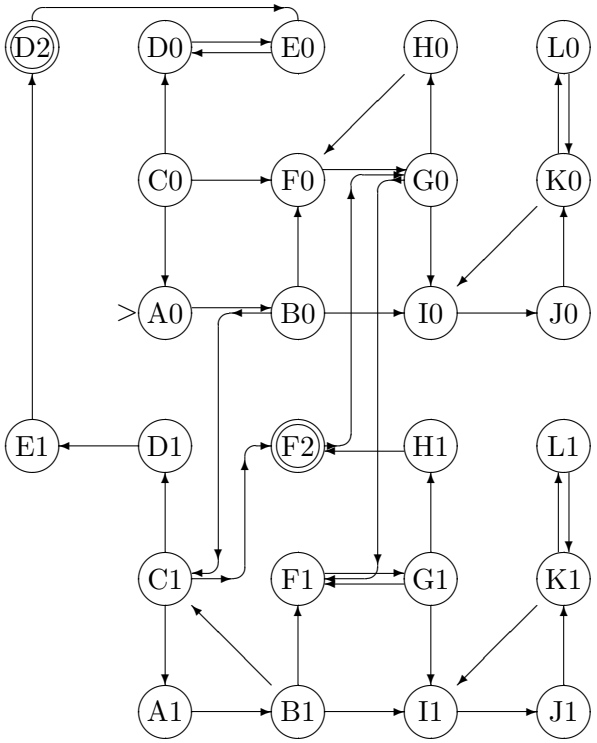


FIG. 2.10 – Automate à transitions développé

Les autres algorithmes [47], [40] fonctionnant à la volée ne traitent que des automates de Büchi (des automates à états avec une unique condition). Ils consistent en deux parcours en profondeur. Dans sa dernière version, l'algorithme dit magique [40], à chaque fois que tous les successeurs d'un sommet acceptant sont traités par le premier parcours, un second parcours est lancé pour tester si ce sommet est accessible par un chemin non nul (voir [40] pour une présentation complète de l'algorithme). Cet algorithme a deux défauts : (1) Il ne traite pas des automates à transitions et nécessite une transformation au préalable de l'automate ; (2) Il ne détecte pas de cycle acceptant dès que le graphe parcouru en contient un (voir l'exemple 7).

Exemple 7 *Pour appliquer l'algorithme magique [40] à l'automate à transitions donné dans la figure 2.8, il faut le transformer en un simple automate de Büchi. Ceci peut être réalisé en le synchronisant avec l'automate compteur de la figure 2.9. La figure 2.10 donne le résultat de cette opération.*

La table 7 donne les états du premier parcours de l'algorithme magique [40] sur l'automate transformé en un automate de Büchi. Le second parcours ne pourra commencer qu'à l'étape 30 : quand le premier parcours aura complètement traité le sommet $F2$*. Le second parcours (non représenté dans la table 7) conduira à la détection du cycle acceptant* $F2 \rightarrow G0 \rightarrow F1 \rightarrow G1 \rightarrow H1 \rightarrow F2$*. Notez que dès l'étape 13, l'algorithme a parcouru tous les arcs contenus dans le*

	Nœeud	Transition	Hash
1	A0		A0
2	A0	A0→B0	A0,B0
3	B0	B0→C1	A0,B0,C1
4	C1	C1→A1	A0,B0,C1,A1
5	A1	A1→B1	A0,B0,C1,A1,B1
6	B1	B1→C1	A0,B0,C1,A1,B1,C1
7	B1	B1→F1	A0,B0,C1,A1,B1,C1,F1
8	F1	F1→G1	A0,B0,C1,A1,B1,C1,F1,G1
9	G1	G1→F1	A0,B0,C1,A1,B1,C1,F1,G1
10	G1	G1→H1	A0,B0,C1,A1,B1,C1,F1,G1, H1
11	H1	H1→F2	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2
12	F2	F2→G0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0
13	G0	G0→F1	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0
14	G0	G0→H0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0
15	H0	H0→F0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0
16	F0	F0→G0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0
17	F0		
18	H0		
19	G0	G0→I0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0,I0
20	I0	I0→J0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0,I0,J0
21	J0	J0→K0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0,I0,J0,K0
22	K0	K0→I0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0,I0,J0,K0
23	K0	K0→L0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0,I0,J0,K0,L0
24	L0	L0→K0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0,I0,J0,K0,L0
25	L0		
26	K0		
27	J0		
28	I0		
29	G0		
30	F2		

TAB. 2.10 – Une exécution de l'algorithme magique

cycle et n'a pas pu détecter le comportement acceptant.

2.4 Vérification de systèmes probabilistes

Cette partie est consacrée à nos travaux [22] sur la vérification d'une formule LTL sur un système probabiliste. Le résultat principal est la conception d'une méthode à base d'automates pour résoudre ce problème, de complexité optimale. Le point clef de la technique est l'exploitation des qualités de non ambiguïté et de séparation des automates produits par les constructions globales.

Cette partie est organisée en 6 sections. Les deux premières sections donnent quelques notions de la théorie de la mesure et la définition d'un système probabiliste. La troisième section donne des propriétés des composantes fortement connexes du produit synchronisé du système et de l'automate de la propriété. La quatrième section établit le résultat principal sur la vérification d'une formule LTL. La cinquième section décrit une méthode d'évaluation de la probabilité qu'une formule LTL soit vraie. Nous terminons cette partie par une étude ex-

périmentale.

2.4.1 Notions de mesure

Une tribu (ou σ-*algèbre*) $\mathcal{A}$ sur un ensemble X est un ensemble de parties de X tel que $\emptyset \in \mathcal{A}$, $\mathcal{A}$ est clos par complémentation et union dénombrable. La paire $(X, \mathcal{A})$ est appelée *espace mesurable*, et les éléments de $\mathcal{A}$ sont appelés *ensembles mesurables*. Etant donnés deux espaces mesurables $(X_1, \mathcal{A}_1)$ et $(X_2, \mathcal{A}_2)$, une fonction $f : X_1 \to X_2$ est appelée *fonction mesurable* si l'image inverse $f^{-1}(Y_2)$ de tout ensemble mesurable Y_2 de X_2 est un ensemble mesurable de X_1.

Une *mesure* μ sur un espace mesurable $(X, \mathcal{A})$ est une fonction de $\mathcal{A}$ dans $\mathbf{R}^+$ telle que (1) $\mu(\emptyset) = 0$, et (2) pour toute famille dénombrable $(A_n)_{n\in\mathcal{N}}$ d'ensembles deux à deux disjoints dans $\mathcal{A}$, $\mu(\bigcup_{n\in\mathcal{N}} A_n) = \sum_{n\in\mathcal{N}} \mu(A_n)$. Une *mesure de probabilité* sur $(X, \mathcal{A})$ est une mesure μ sur $(X, \mathcal{A})$ tel que $\mu(X) = 1$.

Soit Σ un alphabet fini, notons $\mathcal{C}_\Sigma$ l'ensemble des *cylindres* $w \cdot \Sigma^\omega$ avec $w \in \Sigma^*$, et par $\mathcal{B}_\Sigma$ la tribu (sur Σ^ω) engendrée par $\mathcal{C}_\Sigma$. La proposition suivante donne des propriétés élémentaires sur cette tribu.

Proposition 2.11 *Soit Σ un alphabet fini. Pour tout langage $L \subseteq \Sigma^\omega$, les assertions suivantes sont équivalentes*

i) L est mesurable
ii) $K \cdot L$ est mesurable pour tout langage $K \subseteq \Sigma^*$
iii) $w \cdot L$ est mesurable pour tout mot fini $w \in \Sigma^*$
iv) $K^{-1}L$ est mesurable pour tout langage $K \subseteq \Sigma^*$
v) $a^{-1}L$ est mesurable pour toute lettre $a \in \Sigma$.

Une classe importante de fonctions mesurables est obtenue en étendant les fonctions entre deux alphabets. Nous parlerons de ω-*extension*.

Définition 2.12 *Soit Σ_1, Σ_2 deux alphabets finis. Soit f une fonction de $\Sigma_1 \to \Sigma_2$, une* ω-extension $\bar{f}$ *de f est une fonction $\bar{f} : \Sigma_1^\omega \to \Sigma_2^\omega$ définie par $\bar{f}(a_0 a_1 \cdots a_n \cdots) = f(a_0)f(a_1)\cdots f(a_n)\cdots$.*

Une ω-extension d'une fonction f sera généralement notée f, plutôt que $\bar{f}$.

Proposition 2.13 *L'ω-extension de toute fonction $f : \Sigma_1 \to \Sigma_2$ est mesurable.*

Le théorème suivant montre comment déduire une mesure sur la tribu $\mathcal{B}_\Sigma$ à partir d'une fonction réelle positive sur les cylindres.

Théorème 2.14 (Extension d'une mesure) *Soit f une fonction de $\mathcal{C}_\Sigma \to \mathbf{R}^+$, les assertions suivantes sont équivalentes :*

i) $f(w \cdot \Sigma^\omega) = \sum_{a\in\Sigma} f(wa \cdot \Sigma^\omega)$ pour tout $w \in \Sigma^*$
ii) il existe une unique mesure μ sur $(\Sigma^\omega, \mathcal{B}_\Sigma)$ telle que μ et f sont égales sur $\mathcal{C}_\Sigma$.

2.4.2 Système de transitions probabiliste

Par souci de simplification, nous considérons que les systèmes et les automates à transitions sont étiquetés par des lettres d'un alphabet fini. On parlera donc librement de systèmes ou d'automates sur un alphabet Σ. Nous utilisons les notations ${}^\bullet u$ et $u^\bullet$ pour représenter les successeurs et les prédécesseurs d'un état ou d'une transition u ; nous appliquons aussi cette notation à des ensembles d'états et de transitions.

Un *système de transitions probabiliste* (sur Σ) est une structure $M = \langle S, T, \alpha, \beta, \lambda, P_0, P\rangle$ où :

1. $\langle S, T, \alpha, \beta, \lambda\rangle$ est un système de transitions sur Σ, et
2. $P_0 : S \to [0, 1]$ est une *distribution de probabilité initiale*, i.e. $\sum_{s\in S} P_0(s) = 1$,
3. $P : T \to]0, 1]$ est une *fonction de probabilité sur les transitions* satisfaisant $\sum_{t\in s^\bullet} P(t) = 1$, pour tout $s \in S$.

Nous pouvons observer que la troisième condition impose que tout état est source d'au moins une transition.

A partir du théorème 2.14, nous définissons μ_M comme l'unique *mesure* sur $(T^\omega, \mathcal{B}_T)$ vérifiant les identités suivantes sur les cylindres :

1. $\mu_M(T^\omega) = 1$, et
2. pour tout mot non vide $t_0 t_1 \cdots t_n \in T^*$,

$$\mu_M(t_0 t_1 \cdots t_n \cdot T^\omega) = \begin{cases} P_0({}^\bullet t_0)P(t_0)P(t_1)\cdots P(t_n) \text{ si } t_0 t_1 \cdots t_n \in Path^*(M) \\ 0 \text{ sinon.} \end{cases}$$

Dans la suite de notre étude, $(T^\omega, \mathcal{B}_T, \mu_M)$ est considéré comme *espace de probabilité*.

Proposition 2.15 *Nous avons* $\mu_M(Path^\omega(M)) = 1$.

Rappelons qu'un langage défini par un automate à transitions est mesurable :

Théorème 2.16 ([84]) *Pour tout automate à transitions A, le langage $L(A)$ est mesurable sur Σ^ω.*

Nous en déduisons que l'ensemble des comportements infinis (d'un système probabiliste) satisfaisant une formule LTL est mesurable, et ainsi sa probabilité peut être définie

Corollaire 2.17 *Soit A un automate à transitions et M un système probabiliste, l'ensemble des comportements de M "acceptés" par A (i.e. $Path^\omega(M) \cap \lambda^{-1}(L(A))$) est un sous ensemble mesurable de T^ω.*

L'ensemble des états $s \in S$ tel que $P_0(s) > 0$ représente les *états initiaux*. Soit s un état, notons $M[s]$ le système probabiliste M avec s comme unique état initial (i.e. la distribution initial P'_0 de $M[s]$ est définie par $P'_0(s) = 1$). Les observation suivante sont facilement déduites de la propriété d'unicité de la mesure μ_M :

Proposition 2.18 *Pour tout langage mesurable $L \subseteq T^\omega$, nous avons :*

$$\mu_M(L) = \sum_{s \in S_0} P_0(s) \cdot \mu_{M[s]}(L).$$

Proposition 2.19 *Pour tout état s, nous avons $\mu_{M[s]}(Path^\omega(M,s)) = 1$.*

Proposition 2.20 *Pour tout état s et tout langage mesurable $L \subseteq T^\omega$, nous avons :*

$$\mu_{M[s]}(L) = \sum_{t \in s^\bullet} P(t) \cdot \mu_{M[t^\bullet]}(t^{-1}L).$$

Lemme 2.21 *Soit $K \subseteq T^*$. S'il existe $k \in \mathcal{N}$ tel que pour tout $\sigma \in Path^*(M)$, nous avons $\sigma \cdot \sigma' \in K \cdot T^*$ pour au moins un mot $\sigma' \in T^k$, alors nous avons $\mu_M(K \cdot T^\omega) = 1$.*

Les deux propositions suivantes sont déduites du lemme 2.21, et établissent les relations d'équités fortes d'un système probabiliste : (1) un comportement finit inévitablement dans une composante fortement connexe maximale (ou puit), et (2) un comportement aboutissant dans une composante fortement connexe maximale donnée parcourt presque sûrement infiniment souvent tout chemin fini de la composante.

Proposition 2.22 *Soit $Path^*_{max}$ l'ensemble des comportements finissant dans une composante fortement connexe maximale. Nous avons $\mu_M(Path^*_{max} \cdot T^\omega) = 1$.*

Proposition 2.23 *Soit ρ un chemin fini contenu dans une composante fortement connexe maximale C, et soit s un état de C. Nous avons $\mu_{M[s]}((T^* \cdot \rho)^\omega) = 1$.*

2.4.3 Propriétés du produit synchronisé

A partir de maintenant, nous considérerons le système probabiliste $M = \langle S, T_M, \alpha_M, \beta_M, \lambda_M, P_0, P\rangle$ et l'automate à transitions $A = \langle Q, T_A, \alpha_A, \beta_A, \lambda_A, Q_0, Acc\rangle$. Le produit synchronisé du système et de l'automate sera noté $M \otimes A = \langle S \times Q, T_\otimes, \alpha_\otimes, \beta_\otimes, \lambda_\otimes, S_0 \times Q_0, Acc_\otimes\rangle$.

Rappellons que le produit synchronisé $M \otimes A$, le système M et l'automate A sont liés par l'identité

$$L(M \otimes A) = Path^\omega(M) \cap \lambda_M^{-1}(L(A)).$$

Nous voulons déterminer si M satisfait A avec une probabilité positive ($\mu_M(L(M \otimes A)) > 0$). Le langage $L(M \otimes A)$ peut être réécrit comme une union des $L(M \otimes A[s,q])$ où les (s,q) représentent les états initiaux du produit synchronisé. Ainsi, le problème de la vérification probabiliste peut être réduit à tester si $\mu_M(L(M \otimes A[s,q])) > 0$ pour au moins un état $(s,q) \in S_0 \times Q_0$.

Par souci de lisibilité, nous noterons $L(s,q)$, le langage $L(M \otimes A[s,q])$, et nous définissons la fonction $V : S \times Q \to [0, +\infty[$ avec $V(s,q) = \mu_{M[s]}(L(s,q))$.

Remarquez que $L(s,q) = Path^{\omega}(M[s]) \cap \lambda^{-1}(L(A[q]))$. Comme $M \otimes A$ est vu comme un automate à transitions, alors pour tout $(s,q) \in S \times Q$, nous avons :

$$L(s,q) = \bigcup_{(t_M,t_A)\in(s,q)^{\bullet}} t_M \cdot L({t_M}^{\bullet}, {t_A}^{\bullet}).$$

Nous déduisons de l'égalité précédente que $V(s,q) > 0$ ssi $V(s',q') > 0$ pour au moins un état (s',q') accessible à partir de (s,q) (i.e. $(s,q) \xrightarrow{*} (s',q')$). Ainsi, l'ensemble des états $V^{-1}(]0,+\infty[)$ est clos en arrière, et toute SCC (composante fortement connexe) est incluse dans $V^{-1}(]0,+\infty[)$ ou dans $V^{-1}(\{0\})$. Ceci nous amène à une classification des composantes fortement connexes :

Définition 2.24 *Une SCC C de $M \otimes A$ est appelée :*
- nulle *si $C \subseteq V^{-1}(\{0\})$,*
- persistante *si C est une SCC maximale parmis les SCCs non nulles,*
- transitoire *sinon.*

Il en résulte que la vérification probabiliste se réduit à tester l'existence d'une SCC non nulle accessible, ou, de manière équivalente, une SCC persistante accessible. Malheureusement, ces notions ne sont pas *locales* : par exemple, la persistance d'une SCC dépend des autres SCC de $M \otimes A$. Pour concevoir une méthode de vérification efficace, nous recherchons une caractérisation locale des SCC non nulles.

Soit C une SCC de $M \otimes A$ et (s,q) un état de C, notons $(M \otimes A)_{|C}[s,q]$ l'automate $M \otimes A$ restreint aux états de C, nous définissons $L_C(s,q)$ et $V_C(s,q)$ par : $L_C(s,q) = L((M \otimes A)_{|C}[s,q])$ et $V_C(s,q) = \mu_{M[s]}(L_C(s,q))$.

Définition 2.25 *Une SCC C de $M \otimes A$ est dite* localement positive *si $V_C(s,q) > 0$ pour tout $(s,q) \in C$.*

Remarque Une SCC C est localement positive ssi $V_C(s,q) > 0$ pour au moins un état $(s,q) \in C$.

Il est clair qu'une SCC persistante est localement positive. La réciproque est fausse, mais une SCC localement positive est non nulle. Ainsi, nous aboutissons à la proposition suivante :

Proposition 2.26 *Nous avons $\mu_M(L(M \otimes A)) > 0$ ssi il existe une SCC localement positive accessible à partir d'un état initial de $M \otimes A$.*

2.4.4 Problème de la satisfaction

Comme les SCC localement positives jouent un rôle majeur dans la vérification probabiliste, nous en recherchons une caractérisation facilement testable. Cette caractérisation est basée sur deux propriétés des SCC : la *complétude* et l'*acceptation*. Nous montrons qu'une SCC est localement positive ssi elle est complète et acceptée. Nous montrons que ces deux propriétés sont faciles à vérifier quand l'automate A est non ambigu et séparé (ce qui est le cas des automate d'une formule LTL produit par les constructions globales).

Définition 2.27 *Soit C une SCC de $M \otimes A$. Notons $Path^*(M,s)$, l'ensemble des chemins de M partant de l'état s. Notons $Trace^*((M \otimes A)_{|C}, (s,q))$, l'ensemble des traces des chemins finis dans la composante C partant de l'état (s,q). Alors C est dite complète si pour état $s \in \pi_M(C)$, nous avons :*

$$Path^*(M,s) = \bigcup_{q \in Q \mid (s,q) \in C} Trace^*((M \otimes A)_{|C}, (s,q)). \tag{2.1}$$

Remarque Une SCC C est complète ssi (2.1) est satisfaite pour au moins un état $s \in \pi_M(C)$.

Nous dirons qu'une SCC C est acceptée si l'ensemble des transitions de A "contenues" dans C intersecte toutes les conditions d'acceptation de l'automate.

Définition 2.28 *Une SCC C de $M \otimes A$ est* acceptée *si $\forall acc \in Acc_\otimes, ({}^\bullet C \cap C^\bullet) \cap acc \neq \emptyset$.*

Le résultat suivant montre que toute composante complète et acceptée est localement positive. La proposition établit un résultat plus précis, utile dans la méthode d'évaluation de la probabilité d'une formule.

Proposition 2.29 *Soit C une SCC de $M \otimes A$. Si C est complète et acceptée, alors pour tout $s \in \pi_M(C)$ nous avons :*

$$\mu_{M[s]} \left(\bigcup_{q \in Q \mid (s,q) \in C} L_C(s,q) \right) = 1.$$

La proposition suivante établit la réciproque.

Proposition 2.30 *Si C est localement positive de $M \otimes A$, alors C est complète et acceptée.*

Nous avons montré jusqu'à présent que les deux assertions suivantes sont équivalentes :

i) M satisfait A avec une probabilité positive,

ii) il existe une SCC complète et acceptée accessible à partir d'un état initial de $M \otimes A$.

Observons que l'acceptation d'une SCC est extrêmement simple à vérifier. Nous montrons maintenant que la propriété de complétude peut être testée efficacement quand A est non ambigu et séparé. Informellement, sous cette hypothèse, le test de complétude se ramène à compter les transitions de la composante, comme l'exprime le lemme suivant :

Lemme 2.31 *Soit C une SCC de $M \otimes A$. Si A est non ambigu et séparé, alors les deux assertions suivantes sont équivalentes :*

– *C est complète*

– *pour tout* $(s,q) \in C$, s *appartient à une SCC de* M *maximale et*

$$\sum_{s \in \pi_M(C)} | {}^\bullet s \cap SCC(s)^\bullet | \cdot | \pi_M^{-1}(s) \cap C | \quad = \quad | {}^\bullet C \cap C^\bullet | .$$

Les théorèmes suivants sont déduits des propositions précédentes, et de la méthode de construction globale d'un automate pour une formule LTL.

Théorème 2.32 *Si* A *est non ambigü et séparé alors tester que* M *satisfait* A *avec une probabilité non nulle peut être réalisé en temps* $O(|\ Acc\ |) \cdot O(|\ M \otimes A\ |)$.

Théorème 2.33 *Soit* f *une formule LTL. Tester que* M *satisfait* f *avec une probabilité non nulle peut être réalisé en temps* $O(|\ el_T(f)\ |) \cdot O(|\ M\ |) \cdot O(2^{|el_T(f)|})$.

2.4.5 Problème de l'évaluation

Le problème de l'évaluation ne peut être raisonnablement étudié seulement quand l'automate est non ambigu : cette propriété implique que les probabilités $V(s,q)$ sont des solutions d'un système d'équations linéaires. Ce système peut être résolu sur chaque SCC non nulle suivant un ordre topologique. La nature de ces systèmes dépend du type de composantes : pour les composantes transitoires, les systèmes n'ont qu'une solution, pour les composantes persistantes, l'ajout d'une équation indépendante est nécessaire pour résoudre le système. Dans le cas où l'automate A est séparé, nous donnons les équations linéaires manquantes. Notez que ces résultats conduisent à une méthode de calcul de la probabilité d'une propriété quand l'automate est non ambigu et séparé, et tout particulièrement quand la propriété est exprimée par une formule LTL.

Proposition 2.34 *Soit* A *un automate non ambigu. Soit* C *une SCC de* $M \otimes A$. *Soit* E_C *le système d'équations linéaires*

$$V(s,q) = \sum_{(t_M, t_A) \in (s,q)^\bullet} P(t_M) \cdot V({t_M}^\bullet, {t_A}^\bullet)$$

avec $(s,q) \in C$. *Si* C *est persistente alors* $rang(E_C) = |C| - 1$. *Si* C *est transitoire alors* $rang(E_C) = |C|$.

Proposition 2.35 *Si* A *est un automate séparé et* C *est une SCC persistante de* $M \otimes A$ *alors*

$$\sum_{q:(s,q) \in C} V(s,q) = 1$$

2.4.6 Expérimentations

Nous avons implémenté notre technique dans l'outil ProbaTaf. Les systèmes probabilistes sont décrits par des réseaux de Petri (bornés). Les formules LTL sont définies à partir des noms des transitions, de propositions sur les marquages (e. g. "P" pour "la place P est marquée") et la proposition élémentaire *dead*

n		3	4	5	6	7	8	9	10	11	12	13
$X^n dead$	sat	2	2	11	46	99	268	836	1815	4205	9713	19686
	eval	10	24	36	73	161	378	977	2114	5053	9665	21057
$X^n dead$	sat	1	1	2	4	3	6	10	6	10	13	9
$\wedge G(dead \Rightarrow X dead)$	eval	2	6	6	8	11	23	26	32	40	50	56

TAB. 2.11 – Temps de calcul des problèmes sat. et eval. appliqués au jeu de dé

pour les transitions fictives attachées aux états bloquants. L'outil permet aussi bien de tester la satisfaction d'une formule et son évaluation. Nous avons utilisé plusieurs techniques avancées :

- La technologie des BDD pour calculer, simplifier et stocker l'automate traduisant une formule.
- Une adaptation de l'algorithme de vérification à la volée.
- Un algorithme simplifié d'élimination de Gauss pour résoudre les systèmes d'équations linéaires : il est toujours possible de sélectionner les pivots sur la diagonale.

Nous concluons notre étude par l'application de notre technique à deux exemples. L'objectif est de donner un bref aperçu de l'intérêt et des limites de notre approche.

Exemple 8 *Le premier exemple est emprunté à Knuth et Yao[51]. Ils introduisent des questions essentielles relatives à la génération de valeurs aléatoires. La première technique concerne la génération de valeurs aléatoires avec une distribution uniforme sur un ensemble fini. Supposons que l'on veuille simuler un dé parfait avec une pièce de monnaie biaisée. Il suffit de lancer la pièce quatre fois et d'écarter les cas où le nombre de piles et de faces sont différents. Si les lancers sont acceptés, nous utilisons la table de correspondance suivante : 0011 → 1, 0101 → 2, 0110 → 3, 1001 → 4, 1010 → 5, 1100 → 6 (pile est codé par 1 et face par 0). Quand les trois premiers lancers sont identiques, nous pouvons écarter immédiatement les lancers. Le jeu continue jusqu'à ce qu'un résultat soit valide.*

L'outil ProbaTaf nous a permis de vérifier expérimentalement la propriété d'uniformité pour différentes valeurs de biais de la pièce. Cependant, nous avons observé que le nombre de lancers croît en fonction de la valeur de biais. Pour vérifier cette propriété, nous avons considéré la propriété "obtenir un résultat en moins de n lancers". Cette propriété s'exprime en LTL par $X^n dead$ et nous l'avons évaluée pour plusieurs valeurs de biais. Hélas, le problème de satisfaction et d'évaluation de cette formule a un coup exponentiel en fonction de n (voir la table 2.11). En effet, l'automate traduisant la formule LTL par la méthode globale a un nombre exponentiel d'états. L'ajout de la tautologie $G(dead \Rightarrow X dead)$ à notre formule réduit la complexité des calculs à des temps linéaires (voir la table 2.11). Cette expérimentation met en avant le problème d'explosion combinatoire. Ce problème pourrait être partiellement résolu en développant une méthode de construction locale produisant des automates non ambigus et séparés sur leurs composantes fortement connexes.

i	0	1	2	3	4
$F(E_i \wedge X(dead \wedge \neg E_i))$	0.034	0.068	0.068	0.068	0.0010

TAB. 2.12 – Probabilité d'être deuxième sur un arbre équilibré de profondeur 5

$profondeur$		2	3	4	5
$F(dead \wedge E_0)$	sat	3	7	8	4841
	eval	21	19	94	99797
$F(E_0 \wedge X(dead \wedge \neg E_0))$	sat	3	3	19	4894
	eval	17	28	242	336821

TAB. 2.13 – Temps de calcul des problèmes sat. et eval. sur l'algorithme d'élection

Exemple 9 *Dans [62], les auteurs proposent un algorithme distribué probabiliste sur un réseau bidirectionnel acyclique connexe résolvant le problème de l'élection. Le principe de cet algorithme est extrêmement simple : un sommet n'ayant plus qu'un voisin peut être éliminé ; un sommet sans voisin devient alors l'élu. En jouant sur les probabilités qu'un sommet ayant un unique voisin soit éliminé, les sommets du graphe ont la même probabilité d'être élus. L'idée maîtresse est d'associer à tout sommet un poids définissant linéairement sa probabilité d'être éliminé parmi les sommets sans voisin. Initialement, les sommets ont un poids de 1. Quand un sommet est retiré du graphe, il donne son poids à son unique voisin.*

L'outil ProbaTaf nous a permis d'observer la propriété d'uniformité de l'algorithme pour plusieurs exemples de graphes. L'expérimentation consiste à évaluer la probabilité de la formule $F(dead \wedge E_i)$ où E_i indique que le sommet i reste candidat à élection. Par pur hasard, nous avons mis en lumière une propriété surprenante de l'algorithme : la probabilité qu'un sommet donné soit finaliste (élu ou dernier éliminé) est proportionnelle à son nombre de voisins. Nous observons cette propriété dans la table 2.12 sur un arbre équilibré de profondeur 5, où l'index i représente la profondeur du sommet étudié. Depuis cette constatation, un des concepteurs de l'algorithme a prouvé formellement la validité de la propriété.

Nous avons conduit quelques expérimentations sur l'algorithme d'élection pour des arbres équilibrés. La taille du système probabiliste croît doublement exponentiellement en fonction de la profondeur de l'arbre. Pour un arbre de profondeur 5, le système probabiliste à 459829 états et 3599198 transitions. La table 2.13 donne le temps de calcul pour tester la satisfaction et évaluer la probabilité de la formule "être deuxième". La différence des temps de calcul s'explique essentiellement par l'utilisation d'un algorithme à la volée pour résoudre le problème de satisfaction, amplifié par le coût de la résolution des systèmes d'équations linéaires utilisée pour évaluer la probabilité.

2.5 Conclusion

Dans ce chapitre, nous avons abordé trois aspects de la vérification LTL : la traduction d'une formule en un automate à transitions, le test du vide d'un automate à transitions et la vérification de systèmes probabilistes.

Malgré les nombreux travaux récents, nous avons expérimentalement montré que la méthode de construction d'automates que j'ai proposée en 1999, est très compétitive malgré sa simplicité. Ce résultat est à prendre avec beaucoup de précaution. En effet, j'ai profité de la qualité et de l'enthousiasme des concepteurs de l'outil Spot et tous les créateurs de nouvelles méthodes n'ont pas eu une chance équivalente. La réalisation d'un outil efficace exige un module de simplification de formules LTL, un module de construction intégrant des optimisations et un module de simplification d'automates. La moindre faille dans l'implémentation peut rendre l'outil peu compétitif. A mon avis, la qualité première de ma méthode est sa simplicité. Elle n'exige qu'un nombre très restreint d'optimisations pour être efficace et ne nécessite pas de module de simplification d'automates. Mon attention initiale était pédagogique et répondait au besoin de traiter intégralement des exemples de calcul d'automates. Or, même actuellement, peu d'articles n'arrivent à illustrer leurs méthodes par un exemple. La solution que j'ai proposée répondait parfaitement à ces besoins. Pour preuve, mes premières données expérimentales étaient réalisées à la main. Cette qualité est un atout pour ma technique pour qui veut implémenter une méthode de construction d'automates dans le dessein de concevoir un outil de vérification.

Jusqu'à présent, la méthode la plus utilisée pour tester à la volée le vide d'un automate est l'algorithme magique. Nous avons mis en évidence par un exemple que cet algorithme ne fonctionne que partiellement à la volée. Dans un article récent [36], les auteurs ont conçu un algorithme analogue au mien, basé sur l'algorithme de Tarjan. Ils ont montré par de nombreuses expérimentations que leur algorithme parcourt beaucoup moins d'états que l'algorithme magique quand un automate contient un cycle acceptant. Il est plus que probable que mon algorithme possède les mêmes qualités. Il possède un atout supplémentaire : traiter des automates à transitions. Cependant il faut se méfier des conclusions hâtives ; une étude expérimentale doit être menée pour évaluer la qualité de l'algorithme. Avec les concepteurs de l'outil Spot, nous avons le projet de réaliser une version de l'outil Spin basée sur les automates à transitions utilisant les mêmes techniques de réduction par ordre partiel. Nous pensons adapter notre algorithme de vérification pour tenir compte plus finement de certaines propriétés d'équité dans le parcours partiel des états du système étudié. Informellement, dans les techniques de réduction par ordre partiel, une action réalisable du système ne peut être indéfiniment oubliée ; sinon les propriétés de sûreté et de vivacité ne seront pas préservés par le graphe réduit du système. La réalisation de ce projet donnera, à mon avis, un outil fiable d'évaluation des méthodes et des idées que j'ai développées dans ce chapitre.

Nos contributions sur la vérification de systèmes probabilistes sont essentiellement théoriques. En effet, nous avons résolu le problème de la vérification d'une formule LTL par une technique à base d'automates. Pour être efficace, il

est essentiel de concevoir des techniques de traduction locale d'une formule LTL pour produire des automates non ambigus et séparés. Aborder des systèmes de grandes tailles ne pourrait être possible qu'en adaptant des techniques ayant fait leur preuve, telles que les méthodes par ordre partiel. L'aboutissement de ce projet constitue un véritable chalenge

Chapitre 3

Dépliages de réseaux de Petri

Ce chapitre est consacré à la présentation d'un ensemble de méthodes dites par *ordre partiel*. Jusqu'à présent, nous nous sommes intéressés aux algorithmes de base pour la vérification d'une formule LTL. L'objectif des méthodes par *ordre partiel* concerne l'optimisation des méthodes de vérification en tenant compte des notions d'entrelacement et de concurrence des événements d'un système. Dans certaines de ces méthodes [82, 86, 83, 68, 39], le principe est de parcourir partiellement l'espace d'états tout en garantissant la validité de la vérification. Ce sont ces techniques qui ont été mises en œuvre dans des outils comme SPIN [48]. Les méthodes abordées dans ce chapitre sont basées sur une représentation explicite de l' *ordre partiel* liant les événements du système étudié. Le modèle retenu pour représenter les comportements est un type particulier de réseaux de Petri, les *réseaux d'occurrence*. Le réseau d'occurrence décrit l'ordre partiel entre les événements, alors qu'un homomorphisme associe aux événements les transitions du réseau étudié. Cette association est connue dans la littérature sous le terme de processus arborescent. Elle a été initialement introduite dans [66, 27] pour définir une sémantique arborescente de la concurrence dans les réseaux de Petri. Un résultat majeur de cette construction est l'existence d'un unique plus grand processus arborescent contenant tous les processus, appelé *dépliage*.

Pour un réseau de Petri, le nombre d'événements du dépliage est infini dès qu'une séquence infinie de franchissements est réalisable. Les méthodes de vérification que nous présentons ici, reposent sur le calcul d'une partie du dépliage du réseau, appelée *préfixe fini*. Une des difficultés majeures de ces méthodes est de s'assurer qu'un préfixe fini est suffisamment grand pour permettre la vérification d'une propriétés donnée. McMillan [60, 59] donne une méthode de construction d'un préfixe fini adapté à la détection d'états bloquants et plus généralement à la vérification de certaines propriétés d'accessibilité. De nombreux auteurs [28, 20, 69, 41, 21, 18, 29, 30] ont étendu ces méthodes pour la vérification de propriétés de vivacité, et plus généralement de propriétés de logique temporelle.

Dès 1996, nous avons proposé une nouvelle représentation du graphe d'états en termes de graphe de préfixes finis [69, 20, 21, 18] et l'avons appliquée à la vérification de propriétés de logique temporelle. Le principe repose sur une

idée simple : (1) identifier les transitions observables (celles pouvant modifier les valeurs des propositions élémentaires intervenant dans la formule) ; (2) construire un graphe dont les nœuds sont des préfixes finis ne faisant pas apparaître de transitions observables ; (3) réaliser la vérification sur ce graphe en tenant compte des comportements bloquants et cycliques contenus dans les nœuds.

En 2001, nous avons poursuivi nos travaux sur le dépliage selon deux directions. Nous avons généralisé les méthodes de dépliage à des réseaux de Petri intégrant des symétries, et montré comment déplier des produits de réseaux de Petri symétriques. L'aboutissement de cette étude est la conception d'une méthode efficace pour traiter des systèmes communicants par rendez-vous et par échanges de messages [1].

Après avoir introduit les notions de base (section 3.1) sur lesquelles reposent les méthodes de vérification, nous présentons la structure fondamentale qu'est le dépliage d'un réseau de Petri (section 3.2). La suite de ce chapitre est consacrée à l'étude des méthodes de vérification proprement dites. La section 3.3 est dédiée à la construction de préfixes finis, à la vérification de propriétés de sûreté et à l'existence de comportements infinis. La section 3.4 présente de manière synthétique nos travaux sur les graphes de préfixes fini et leurs applications à la vérification de formules de logique temporelle. Les sections 3.5, 3.6 et 3.7 traitent du dépliage de réseaux symétriques et de produit de réseaux symétriques.

3.1 Notions élémentaires

Cette section est dédiée à la présentation des deux notions essentielles sur lesquelles reposent la méthode des dépliages :

- les *homomorphismes de réseaux* qui précisent comment les comportements d'un réseau peuvent être simulés par un autre réseau,
- les *réseaux d'occurrence* qui sont une classe particulière de réseaux de Petri employée pour cette simulation.

3.1.1 Réseaux de Petri

La notion de *multi-ensemble* est une structure algébrique qui permet de définir des réseaux de Petri avec un nombre infini de places et de transitions.

Définition 3.1 (Multi-ensemble) *Soit E un ensemble (fini ou infini). Une application u de E dans $\mathbb{N}$ est un* multi-ensemble *si u est à support fini ($u^{-1}(\mathbb{N}^*)$ est fini). L'ensemble des multi-ensembles de E est noté $\mathbb{N}^{|E|}$.*

Un multi-ensemble sur un ensemble E est généralement représenté par une combinaison linéaire fini d'éléments de E. Nous serons amenés à considérer les extensions suivantes sur les multi-ensembles :

- Tout sous-ensemble fini X de E est considéré comme un multi-ensemble de E : $X = \sum_{x \in X} x$.

– Soient E et F deux ensembles. Toute application h de E dans F est étendue en une application sur les multi-ensembles de E dans ceux de F : $\forall u \in \mathbb{N}^{|E|}, h(u) = \sum_{e \in E} u(e) \cdot h(e)$.

Les réseaux considérés dans ce chapitre ne sont pas nécessairement finis, i.e., les nombres de places et/ou de transitions peuvent être infinis. Toutefois, nous imposons que pour toute transition t, Pre(t) et Post(t) ont un support fini et non nul, et que le marquage initial est aussi un support fini.

Définition 3.2 (Réseau de Petri) *Un réseau de Petri R est une structure $\langle P, T, Pre, Post \rangle$ où*

– *P et T sont des ensembles (finis ou infinis) disjoints,*

– *Pre et Post sont de fonctions de T dans $\mathbb{N}^{|P|} \setminus \{0\}$.*

Les éléments de P et T sont appelés des places et des transitions, et Pre et Post sont les fonctions de pré- et post-conditions des transitions. Un marquage *m de R est un multi-ensemble de P ($m \in \mathbb{N}^{|P|}$). Un réseau marqué $\langle R, m_0 \rangle$ est la donnée d'un réseau de Petri R et d'un marquage initial m_0. Un réseau étiqueté $\langle R, \Sigma, \lambda \rangle$ est la donnée d'un réseau R, d'un alphabet fini Σ et d'une fonction d'étiquetage sur les transitions $\lambda : T \to \Sigma$.*

La sémantique opérationnelle d'un réseau de Petri est donnée par la règle de franchissement.

Définition 3.3 (Règle de franchissement) *Soit $R = \langle P, T, Pre, Post \rangle$ un réseau de Petri. Soient t une transition et m un marquage de R. Une transition t est franchissable à partir de m (noté par $m[t >$) si $Pre(t) \leq m$. Franchir une transition t à partir du marquage m amène au marquage m' (noté par $m[t > m'$) défini par :*

$$m' = m + Post(t) - Pre(t). \tag{3.1}$$

La relation de franchissement peut être étendue inductivement à tout séquense de transitions par : $m_1[\epsilon > m_2$ si $m_1 = m_2$, et $m_1[\sigma \cdot t > m_2$ si $\exists m : m_1[\sigma > m \wedge m[t > m_2$. Un marquage m est accessible à partir d'un marquage m_0 (ou accessible dans le réseau marqué $\langle R, m_0 \rangle$) si $\exists \sigma \in T^* : m_0[\sigma > m$. Posons $\overline{\sigma}$, l'image commutative de σ, l'état m' atteint à partir de m par le franchissement de σ est donné par l'équation :

$$m' = m + \mathrm{Post}(\overline{\sigma}) - \mathrm{Pre}(\overline{\sigma}). \tag{3.2}$$

Nous notons $Reach(R, m_0)$ l'ensemble des marquages accessibles du réseau marqué $\langle R, m_0 \rangle$. La définition suivante liste des qualités importantes d'un réseau de Petri.

Définition 3.4 (Propriété d'un réseau) *Soit $R = \langle P, T, Pre, Post \rangle$ un réseau de Petri. Un marquage m est dit sain si $\forall p \in P, m(p) \leq 1$. Le réseau de Petri R est dit :*

– ***fini*** *si $|P| < +\infty \wedge |T| < +\infty$,*

– ***élementaire*** *si $\forall p \in P, \forall t \in T, Pre(t)(p) \leq 1 \wedge Post(t)(p) \leq 1$.*

Soit m_0 un marquage de R. Le réseau marqué $\langle R, m_0 \rangle$ est dit :

- ***borné*** *si* $|Reach(R, m_0)| < +\infty$,
- ***sain*** *si* $\forall m \in Reach(R, m_0) : m$ *est sain,*
- ***quasi-vivant*** *si* $\forall t \in T, \exists m \in Reach(R, m_0) : m[t >$.

Dans la suite de ce chapître, nous utilisons les notations suivantes sur la structure graphique d'un réseau de Petri $R = \langle P, T, \mathrm{Pre}, \mathrm{Post}\rangle$:

- l'ensemble des prédécesseurs (resp. des successeurs) d'une place p est défini par : ${}^\bullet p = \{t \in T | \mathrm{Post}(t)(p) \neq 0\}$ (resp. $p^\bullet = \{t \in T | \mathrm{Pre}(t)(p) \neq 0\}$),
- l'ensemble des prédécesseurs (resp. des successeurs) d'une transition t est défini par : ${}^\bullet t = \{p \in P | \mathrm{Pre}(t)(p) \neq 0\}$ (resp. $t^\bullet = \{p \in P | \mathrm{Post}(t)(p) \neq 0\}$),
- Si s est un élément de $P \cup T$, alors ${}^*s$ (resp. s^*) désigne l'ensemble de ses prédécesseurs (resp. successeurs) directs et indirects. ${}^*s$ et s^* sont définis par induction de la façon suivante :
  - $s \in {}^*s \wedge \forall r \in P \cup T, r^\bullet \cap {}^*s \neq \emptyset \Rightarrow r \in {}^*s$
  - $s \in s^* \wedge \forall r \in P \cup T, {}^\bullet r \cap s^* \neq \emptyset \Rightarrow r \in s^*$

Ces dernières notations sont étendues de manière usuelle aux sous-ensembles de nœuds.

3.1.2 Homomorphisme de réseaux

La notion d'*homomorphisme de réseaux* précise comment un réseau peut être (partiellement) simulé par un autre.

Définition 3.5 (Homomorphisme) *Soient* $\langle R_1, m_{01}\rangle$ *et* $\langle R_2, m_{02}\rangle$ *deux réseaux marqués, avec* $R_i = \langle P_i, T_i, Pre_i, Post_i\rangle$ *pour* $i = 1, 2$. *Soit une application* $h : P_1 \cup T_1 \mapsto P_2 \cup T_2$ *telle que* $h(P_1) \subseteq P_2$, $h(T_1) \subseteq T_2$. *h est un* homomorphisme *de* $\langle R_1, m_{01}\rangle$ *vers* $\langle R_2, m_{02}\rangle$ *si*

- $\forall t_1 \in T_1 : Pre_2(h(t_1)) = h(Pre_1(t_1))$,
- $\forall t_1 \in T_1 : Post_2(h(t_1)) = h(Post_1(t_1))$,
- $m_{02} = h(m_{01})$.

En tout premier lieu, la définition impose que le type des nœuds (place ou transition) soit préservé par l'application h. Les deux premières conditions de la définition assurent que l'environnement des transitions soit, lui aussi, préservé par l'application h alors que la troisième impose que les marquages initiaux des deux réseaux correspondent.

Exemple 10 *La figure 3.1 présente deux réseaux de Petri marqués (*R1 *et* R2*) et un homomorphisme de* R1 *vers* R2. *Les nœuds du réseaux* R1 *sont étiquetés (en italique) par leurs images.*

La proposition suivante clarifie la relation entre les comportements de réseaux de Petri à travers un homomorphisme.

Proposition 3.6 (Préservation des comportements) *Soient deux réseaux marqués* $\langle R_1, m_{01}\rangle$ *et* $\langle R_2, m_{02}\rangle$. *Soit* $h : \langle R_1, m_{01}\rangle \mapsto \langle R_2, m_{02}\rangle$ *un homomorphisme. Soient* $m_1, {m_1}'$ *deux marquages de* R_1 *et* t_1 *une transition de* R_1 *tels que* $m_1[t_1\rangle_{R_1} {m_1}'$. *Alors, dans* R_2, $h(m_1)[h(t_1)\rangle_{R_2} h({m_1}')$.

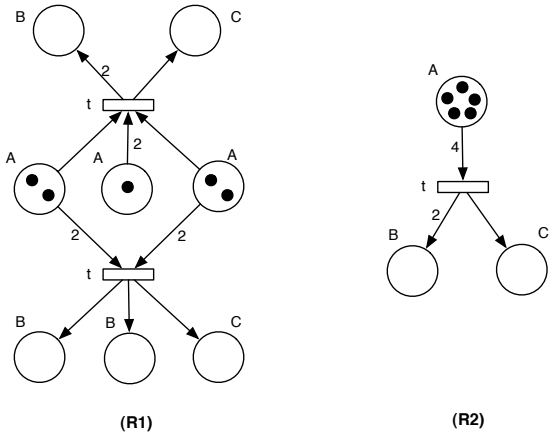


FIG. 3.1 – Deux réseaux Petri marqués et un homomorphisme de $R1$ vers $R2$

La proposition 3.6 peut être étendue par induction aux séquences de transitions. Une conséquence immédiate de cette proposition est que les ensembles de marquages accessibles des deux réseaux sont liés par la propriété :

$$h(Reach(R_1, m_{01})) \subseteq Reach(R_2, m_{02})$$

3.1.3 Réseau d'occurrence

Les réseaux de Petri employés pour définir une sémantique de la concurrence ont une structure particulière. Cette classe de réseaux correspond aux *réseaux d'occurrence* [66].

Définition 3.7 (Réseau d'occurrence) *Un réseau marqué $\langle R, m_0 \rangle$ est un* réseau d'occurrence *si*
- *R est un réseau élémentaire,*
- *$\forall p \in P : |{}^\bullet p| + m_0(p) = 1$*
- *$\langle R, m_0 \rangle$ est quasi-vivant.*

La structure graphique d'un réseau d'occurrence détermine complètement son marquage initial. On désignera un réseau d'occurrence $\langle R, m_0 \rangle$ uniquement par R. Lorsque cela est nécessaire, le marquage initial est noté par $Min(R)$.

La proposition suivante donne les propriétés graphiques fondamentales des réseaux d'occurrence.

Proposition 3.8 (Acyclique) *Le graphe d'un réseau d'occurrence est un DAG (un graphe orienté sans circuit). Pour tout $x \in P \cup T$, l'ensemble ${}^*x$ est fini.*

Pour être un bon candidat, une transition d'un réseau d'occurrence ne doit pas pouvoir être franchie dans une séquence plus d'une fois. En effet, les transitions ont pour vocation de représenter les événements du réseau de Petri étudié. Un rôle dual est affecté aux places du réseau d'occurrence qui doivent représenter la présence de jetons. En conséquence, il est requis que les réseaux d'occurrence soient sains.

Proposition 3.9 (Sain) *Un réseau d'occurrence est un réseau sain. De plus, toute transition peut être franchie au plus une fois dans une séquence de franchissements.*

La structure acyclique du graphe d'un réseau d'occurrence induit des relations sur les places et les transitions.

Définition 3.10 (Causalité, conflit et concurrence) *Soient R un réseau d'occurrence et $x, y \in P \cup T$.*

- *x précède y (noté $x \leq y$) si $x \in {}^*y$,*
- *x et y sont en conflit (noté $x \sharp y$) s'il existe deux transitions distinctes t_x et t_y telles que $t_x \leq x \wedge t_y \leq y \wedge {}^\bullet t_x \cap {}^\bullet t_y \neq \emptyset$*
- *x et y sont concurrents (noté $x || y$) si $\neg(x \leq y)$, $\neg(y \leq x)$ et $\neg(x \sharp y)$.*

Intuitivement, la relation de causalité entre deux transitions $x \leq y$ indique que la transition y ne peut être franchie que si x l'a été précédemment ; la relation de conflit $\sharp$ caractérise les transitions ne pouvant être franchies dans une même séquence de transitions ; et la relation de concurrence $||$ définit des transitions pouvant être exécutées simultanément. Des interprétations analogues s'appliquent aux relations sur les places. Par exemple, si une place est marquée au cours d'une séquence, ses places en conflit resteront vides pendant le franchissement de la même séquence.

Les comportements possibles d'un réseau d'occurrence sont capturés par la notion de *configuration*.

Définition 3.11 (Configuration) *Soit R un réseau d'occurrence. Une configuration C de R est un sous-ensemble de transitions n'étant pas en conflit deux à deux, et clos par le bas (${}^*C \cap T = C$).*

On note $Conf(R)$ l'ensemble des configurations du réseau d'occurrence R. La proposition suivante démontre qu'une configuration est bien une représentation des comportements possibles du réseau considéré et donc qu'à toute séquence correspond une configuration.

Proposition 3.12 (Comportement) *Soient R un réseau d'occurrence et C un sous-ensemble de T. C est une configuration de R ssi $\exists \sigma \in T^*$ telle que $Min(R)[\sigma\rangle \wedge C = \overline{\sigma}$. De plus, le marquage atteint par le franchissement de σ est $Min(R) + C^\bullet - {}^\bullet C$. On note $Cut(C)$ ce marquage.*

Par définition, les transitions composant une configuration ne peuvent être en conflit et sont donc liées soit par la relation de causalité, soit par la relation de concurrence. Ces relations nous permettent de caractériser les séquences de franchissements représentées par une configuration. Une première proposition exploite la relation de causalité pour caractériser l'ordre de franchissement des transitions concernées.

Proposition 3.13 (Causalité) *Soient R un réseau d'occurrence et $\sigma = t_1 \cdots t_n$ une séquence de transitions telle que $\overline{\sigma}$ est une configuration. σ est une séquence de franchissements de R ssi $\forall i, j \in [1, n], t_i \leq t_j \Rightarrow i \leq j$.*

De même, dans une séquence deux transitions voisines et concurrentes peuvent être permutées.

Proposition 3.14 (Concurrence) *Soient R un réseau d'occurrence et $\sigma = t_1 \cdots t_n$ une séquence de franchissements de R. Alors pour tout $i \in [1, n[$,on a $t_i \| t_{i+1} \Rightarrow Min(R)[t_1 \cdots t_{i+1} t_i \cdots t_n\rangle$.*

Ainsi, une configuration définit un ensemble de séquences de franchissements. La propriété de quasi-vivacité des réseaux d'occurrence peut être réécrite en termes d'auto-conflit.

Proposition 3.15 (Non auto-conflit) *Toute transition d'un réseau d'occurrence n'est pas en conflit avec elle-même.*

En conséquence, pour toute transition t, il existe au moins une séquence de franchissements contenant une occurrence de t et donc une configuration correspondante. La proposition suivante caractérise la plus petite (au sens de l'inclusion) de ces configurations.

Proposition 3.16 (Configuration locale) *Soit t une transition d'un réseau d'occurrence R. L'ensemble $({}^*t \cap T)$ est une configuration de R. De plus, cette configuration est la plus petite des configurations de R contenant t. La configuration $({}^*t \cap T)$, noté $[t]$, est appelée la configuration locale de t.*

En termes de vérification, il est souvent nécessaire de déterminer si un marquage partiel peut être couvert par un marquage accessible. La proposition suivante exploite la notion de concurrence pour caractériser les séquences conduisant à un tel marquage couvrant.

Proposition 3.17 (Couverture) *Soient R un réseau d'occurrence et A un ensemble de places deux à deux concurrentes. L'ensemble ${}^*A \cap T$, noté $[A]$, est la plus petite configuration qui couvre A. Si A est réduit à une place, $[p]$ est appelée la configuration locale de p.*

3.2 Processus arborescent et dépliage

Les réseaux d'occurrence associés aux homomorphismes de réseaux permettent la représentation explicite de l'ordre partiel portant sur les événements d'un réseau de Petri. La notion de *processus arborescent* concrétise cette association. Dans cette section, nous montrons que l'ensemble des comportements est représentable dans un unique processus arborescent. Une telle structure est appelé un dépliage.

Dans la suite de ce chapitre, nous considèrons un réseau marqué $\langle R, m_0\rangle$. Les composantes d'un réseau d'occurrence sont traditionnellement notées par $\langle B, E, In, Out\rangle$.

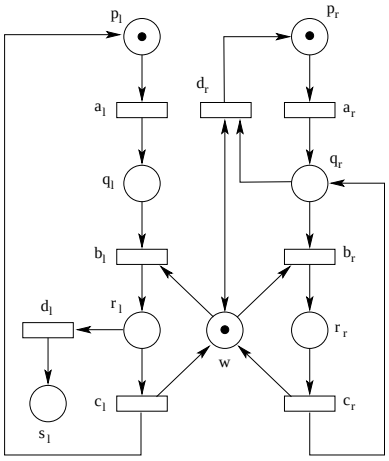


FIG. 3.2 – Un réseau de Petri marqué

3.2.1 Processus arborescents

Un processus arborescent d'un réseau R est la donnée d'un réseau d'occurrence S et d'un homomorphisme h de S dans R. Notons que le réseau S simule partiellement les comportements de R grâce à l'homomorphisme.

Définition 3.18 (Processus arborescent) *Soit $S = \langle B, E, In, Out\rangle$ un réseau d'occurrence. Soit h un homomorphisme de réseau de S dans $\langle R, m_0\rangle$. La paire $\langle S, h\rangle$ est un processus arborescent de $\langle R, m_0\rangle$ si*

$$\forall e_1, e_2 \in E : (In(e_1) = In(e_2) \wedge h(e_1) = h(e_2)) \Rightarrow e_1 = e_2$$

Les éléments des ensembles B et E sont appelés respectivement des conditions et des événements du processus arborescent. La définition 3.18 impose qu'un même événement ne soit représenté qu'une fois au sein d'un même processus arborescent.

Exemple 11 *Les figures 3.2 et 3.3 présentent respectivement un réseau de Petri marqué et un processus arborescent de ce réseau. Les noms des événements et des conditions sont écrits en gras alors que leurs images par l'homomorphisme sont écrit en italique.*

La propriété suivante est essentielle. En effet, elle énonce que toute séquence de franchissements d'un réseau marqué peut être simulée (représentée) par un processus arborescent.

Proposition 3.19 (Séquence de franchissements) *Soit σ une séquence de franchissements de R. Il existe un processus arborescent $\langle S, h\rangle$ de $\langle R, m_0\rangle$ et une séquence de franchissements σ_S de S telle que $h(\sigma_S) = \sigma$.*

Exemple 12 *La séquence $a_r \cdot b_r \cdot c_r \cdot d_r$ est franchissable dans le réseau de la figure 3.2 et est représentée par la séquence $e_2 \cdot e_4 \cdot e_8 \cdot e_{11}$ dans le processus arborescent de la figure 3.3.*

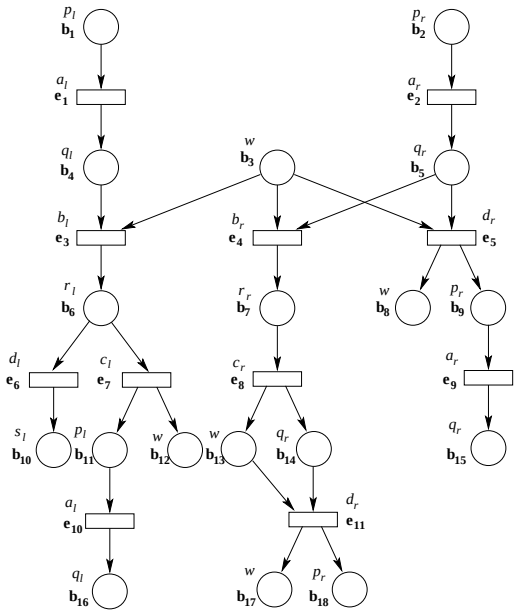
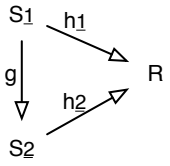


FIG. 3.3 – Un processus arborescent du réseau de la figure 3.2

3.2.2 Dépliages

Après avoir démontré qu'une séquence peut toujours être représentée par un processus arborescent, nous énoncerons l'existence d'un plus grand processus arborescent, appelé *dépliage*. Pour se faire, nous devons avoir la capacité de comparer les processus arborescents entre eux. Cette comparaison est rendue possible par la notion d'homomorphisme de processus arborescents.

Définition 3.20 (Homomorphisme de processus arborescents) *Soient $\langle S_1, h_1\rangle$ et $\langle S_2, h_2\rangle$ deux processus arborescents de $\langle R, m_0\rangle$. Un homomorphisme $g : S_1 \mapsto S_2$ est un homomorphisme de processus arborescents de $\langle S_1, h_1\rangle$ vers $\langle S_2, h_2\rangle$ si $h_1 = h_2 \circ g$.*



Deux processus arborescents sont équivalents s'ils peuvent être reliés par un isomorphisme (i.e. un homomorphisme bijectif) ; dans ce cas, les deux processus arborescents simulent exactement les mêmes comportements.

Définition 3.21 (Relation d'équivalence) *Soient β_1, β_2 deux processus arborescents de $\langle R, m_0\rangle$. On dit que β_1 et β_2 sont équivalents (noté $\beta_1 \equiv \beta_2$) s'il existe un isomorphisme de β_1 vers β_2.*

Un processus arborescent est plus petit qu'un autre s'il peut être obtenu à partir du second en éliminant des événements et des conditions ; ainsi un processus arborescent représentera moins de comportements qu'un processus plus grand que lui. Nous dirons que ce processus est un préfixe de celui qui lui est supérieur. Pour que la relation d'ordre reste compatible par rapport à la relation d'équivalence, cette notion est formalisée en reliant les deux processus par un homomorphisme injectif.

Définition 3.22 (Ordre partiel) *Soient β_1, β_2 deux processus arborescents de $\langle R, m_0\rangle$. β_1 est plus petit que β_2 (noté $\beta_1 \sqsubseteq \beta_2$) s'il existe un homomorphisme injectif de β_1 vers β_2.*

Nous sommes maintenant en mesure d'énoncer la proposition sur l'existence d'un processus arborescent représentant l'ensemble des comportements possibles d'un réseau. C'est un processus arborescent maximal au sens où il inclut (en termes de préfixes) tous les processus arborescents.

Proposition 3.23 (Dépliage) *Soit $\langle R, m_0\rangle$ un réseau marqué. Il existe un unique (à équivalence prêt) plus grand (au sens de $\sqsubseteq$) processus arborescent de $\langle R, m_0\rangle$. Ce processus arborescent est appelé le* dépliage *de $\langle R, m_0\rangle$.*

Tous les processus arborescents d'un réseau sont donc des préfixes de son dépliage. Remarquons que si un réseau peut réaliser une séquence infinie, le dépliage est infini.

3.3 Préfixes finis

Du point de vue de la vérification, il n'est pas souhaitable d'avoir à manipuler une structure potentiellement infinie. Le principe général consiste à ne considérer qu'un préfixe fini du dépliage du réseau. Le problème essentiel est que toutes les informations pertinentes pour la vérification doivent être contenues dans ce préfixe.

Dans cette section, $\langle R, m_0\rangle$ est un réseau marqué fini et $\langle S, h\rangle$ est son dépliage avec $S = \langle B, C, \in, \mathrm{Out}\rangle$.

3.3.1 Définition

Un préfixe fini du dépliage est défini par l'ensemble des événements à partir desquels le dépliage n'est plus considéré. De tels événements sont appelés *raccourci* du préfixe fini.

Définition 3.24 (Préfixe fini) *Un préfixe fini Cutoff de $\langle S, h\rangle$ est un sous-ensemble d'événements de S satisfaisant*

- *$E \setminus Cutoff^*$ est fini,*
- *pour tout $e, e' \in Cutoff, e \not\leq e'$.*

La première contrainte garantit que le préfixe est fini (i.e. il ne contient qu'un nombre fini de conditions et d'événements). La deuxième contrainte impose qu'aucun raccourci n'est jamais causalement après un autre raccourci. Nous définissons sur le préfixe $Cutoff$, les ensembles suivants :

– **L'ensemble des événements (internes)** :

$$Event(Cutoff) = E \setminus Cutoff^*$$

– **L'ensemble des configurations** :

$$Conf(Cutoff) = \{C \in Conf(S) \mid C \cap Cutoff = \emptyset\}$$

– **L'ensemble des marquages** :

$$Reach(Cutoff) = \{h(Cut(C)) \mid C \in Conf(Cutoff)\}$$

Il est important de noter que $Conf(Cutoff)$ ainsi que $Reach(Cutoff)$ sont des ensembles finis. Dans le cadre d'une méthode de vérification, il est courant d'imposer que l'ensemble des états accessibles du réseau étudié soit représenté au sein du préfixe fini. Un tel préfixe est dit *complet*.

Définition 3.25 (Préfixe fini complet) *Un préfixe fini $Cutoff$ de $\langle S, h\rangle$ est complet si $Reach(Cutoff) = Reach(R, m_0)$.*

3.3.2 Ordres adéquats et préfixes finis complets

Il s'agit à présent de définir les conditions suffisantes garantissant la complétude d'un préfixe. Ces conditions vont porter sur les raccourcis délimitant le préfixe fini. Une approche intuitive consiste à s'assurer que les états atteints après le franchissement de la configuration locale d'un raccourci est déjà un état représenté dans le préfixe. Formellement, cette contrainte est définie pour un préfixe $Cutoff$ par

$$\forall e \in Cutoff, \exists C \in Conf(Cutoff) : h(Cut([e])) = h(Cut(C))$$

Dès les premiers travaux sur les préfixes finis, McMillan [60] a mis en évidence que cette condition n'était pas suffisante. L'exemple 13, inspiré de la démonstration de McMillan, illustre ce fait.

Exemple 13 *La figure 3.4 présente un réseau marqué et la figure 3.5 donne une partie du dépliage. Le préfixe fini $Cutoff = \{e_6, e_8\}$ respecte la contrainte donnée ci-dessus. En effet, nous avons*

– $h(Cut[e_6]) = h(\{e_1, e_2, e_5\}) = a_3 + b_2 + c_2 + d_3$
– $h(Cut[e_8]) = h(\{e_3, e_4, e_7\}) = a_2 + b_3 + c_3 + b_2$

Or, bien qu'accessible, le marquage $a_3+b_3+c_3+d_3$ n'appartient pas à $Reach(Cutoff)$. Par contre, les préfixes $Cutoff' = \{e_5, e_8\}$ et $Cutoff'' = \{e_6, e_7\}$ sont bien complets.

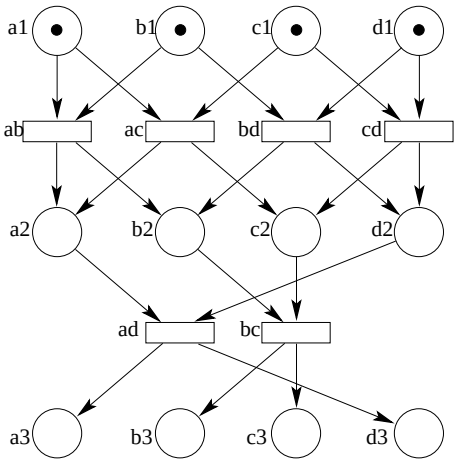


FIG. 3.4 – Contre exemple de complétude

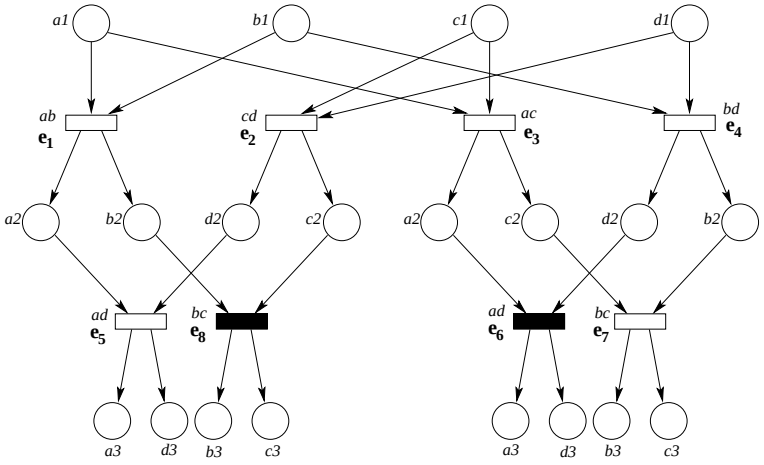


FIG. 3.5 – Un préfixe fini incomplet du contre exemple

Le bon choix des raccourcis d'un préfixe est essentiel pour assurer la complétude. Dans [32], les auteurs proposent une condition suffisante de complétude capturant la plupart des techniques de construction de préfixes complets. Celle-ci est basée sur une classe d'ordres partiels sur les configurations du dépliage, appelée *ordre adéquat*. Cette relation utilise la notion d'extension d'une configuration.

Définition 3.26 (Extension) *Soient C une configuration de S, t une transition de R et e_t un événement de S d'étiquette t (i.e $h(e_t) = t$). L'ensemble $C \cup \{e_t\}$ est une t-extension de C si $C \cup \{e_t\}$ est une configuration et e_t n'est pas un événement de C. L'ensemble des t-extensions de la configuration C est noté $C \cdot t$.*

Définition 3.27 (Ordre adéquat) *[32] Un ordre partiel $\preceq$ sur les configurations d'un dépliage $\langle S, h\rangle$ est un ordre adéquat si :*

- *$\preceq$ raffine $\subseteq$,*
- *$\preceq$ est bien fondé (i.e. il n'existe pas de suite infinie strictement décroissante),*
- *$\preceq$ est compatible avec l'extension : pour toute transition t, pour toutes configurations C_1, C_2 de S telles que $h(Cut(C_1)) = h(Cut(C_2))$*

$$C_1 \prec C_2 \Rightarrow \forall C_2' \in C_2 \cdot t, \exists C_1' \in C_1 \cdot t : C_1' \prec C_2'$$

Dans [60], une configuration est plus petite qu'une autre si elle est composée de moins d'événements. Il est facile de démontrer que cet ordre est un ordre adéquat. Dans [32] est proposé un ordre adéquat total sur les configurations des dépliages de réseaux sains.

La définition et la proposition suivante établit une condition suffisantes pour un préfixe fini soit complet.

Définition 3.28 (*Préfixe fini adéquat*) *Un préfixe fini Cutoff de $\langle S, h\rangle$ est adéquat s'il existe un ordre adéquat $\preceq$ et une application $\phi : Cutoff \mapsto Conf(Cutoff)$ tels que $\forall e \in Cutoff : h(Cut(\phi(e))) = h(Cut([e])) \wedge (\phi(e) \prec [e])$.*

Proposition 3.29 (Complétude d'un préfixe fini adéquat) *Si un préfixe fini est adéquat alors il est un préfixe fini complet. De plus, si $\langle R, m_0\rangle$ est borné alors il existe au moins un préfixe fini adéquat pour tout ordre adéquat.*

Il est important de noter qu'il est toujours possible de construire un préfixe fini adéquat pour tout réseau de Petri borné. Cependant, dans le pire des cas, ce préfixe peut contenir plus d'événements que de marquages accessibles. En revanche, si la relation d'ordre est totale, le nombre d'événements est borné par le nombre de marquages accessibles.

Exemple 14 *Considérons le processus arborescent de la figure 3.3 en tant que partie initiale du dépliage du réseau de la figure 3.2. Le préfixe fini $\{e_5, e_7, e_8\}$ est adéquat pour l'ordre de McMillan (i.e. $C \prec C' \Rightarrow |C| < |C'|$). En effet, la fonction Φ est donnée par : $\Phi(e_5) = \Phi(e_7) = \emptyset$ et $\Phi(e_8) = [e_2]$. Notons que $h(Cut(\Phi(e_5))) = h(Cut(\Phi(e_7))) = h(Cut(\emptyset)) = p_l + w + p_r$ et $h(Cut(\Phi(e_8))) = p_l + w + q_r$. Il est donc complet.*

3.3.3 Vérification de propriété de sûreté

Le principe général de l'algorithme de construction d'un préfixe fini complet est simple. Initialement, une condition est produite pour chaque jeton du marquage initial et les événements pouvant étendre le préfixe sont calculés et stockés dans une liste. A chaque étape du calcul, un événement de plus petite configuration locale (par rapport à l'ordre adéquat choisi) est retiré de la liste, et est ajouté au préfixe. Soit cet événement est défini comme un raccourci, soit cet ajout induit la création de nouveaux événements à ajouter dans liste des événements à traiter. Lorsque cette liste est vide, le préfixe obtenu est un préfixe complet. Le lecteur intéressé pourra trouver dans [72] une description détaillée d'une implémentation efficace de cet l'algorithme.

Beaucoup de propriétés de sûreté peuvent être vérifiées à partir d'un préfixe complet. La détection de la présence d'un état bloquant a été l'une des premières études réalisées sur le sujet [60]. Le principe de détection consiste à construire une configuration $C \in \mathit{Conf}(\mathit{Cutoff})$ telle que pour tout raccourci e d'un préfixe fini adéquat Cutoff, il existe un élément de C en conflit avec e. De fait, cette configuration ne pourra pas être étendue jusqu'à atteindre un raccourci et conduira donc vers un état bloquant. Une définition précise de cet algorithme et des techniques alternatives sont présentées dans [61, 45].

Exemple 15 *Si l'on considère le préfixe fini* $\{e_5, e_7, e_8\}$ *du processus arborescent de la figure 3.3. Nous avons vu que ce préfixe fini est adéquat. L'ensemble* $\{e_1, e_3, e_6\}$ *est une configuration de ce préfixe fini tel que tout raccourci est en conflit avec au moins un événement de la configuration (*$e_3 \sharp e_8, e_3 \sharp e_5$ *et* $e_6 \sharp e_7$*). Cette configuration conduit inévitablement à un état bloquant du réseau (*$s_l + q_r$*).*

La vérification de toutes propriétés de sûreté pouvant être ramenées à un problème de couverture peut être réalisée très simplement à partir d'un préfixe complet. En effet, décider si un marquage partiel peut être couvert revient à décider de la quasi vivacité d'une transition. Il suffit donc d'ajouter au réseau considéré une transition t ayant comme pré-condition $\mathrm{Pre}(t)$, le marquage à couvrir. Si le préfixe complet peut être étendu par un événement étiqueté par cette transition alors cela indique que le marquage peut être couvert. La vérification de propriétés de sûreté ne pouvant pas être réduites à un simple problème de couverture peut être résolu en représentant explicitement les places complémentaires. Ceci impose que la borne de chacune des places soit connue *a priori*. Dans [41] est discuté d'une méthode générale de vérification de propriétés d'accessibilité à partir d'un préfixe fini.

3.3.4 Détection de comportements infinis

La vérification de propriétés de vivacité induit la détection de comportements infinis. Ce problème est difficile en général : dans [87], l'auteur et les référés se sont laissés piéger par l'apparente simplicité du problème [21]. Dans le cas général, deux graphes, dont les nœuds sont les raccourcis et leurs images, jouent un rôle important dans la détection de comportements infinis :

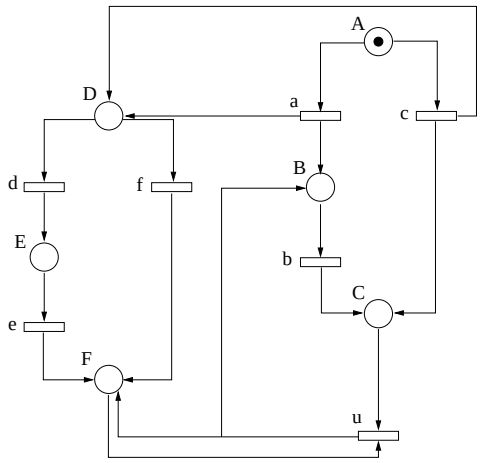


FIG. 3.6 – Premier contre exemple de détection de circuit

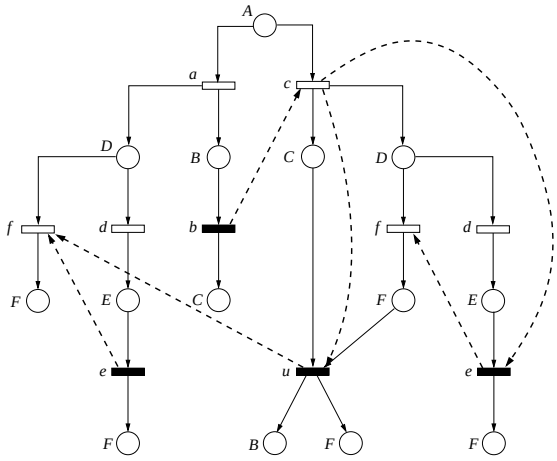


FIG. 3.7 – Non détection de séquence infinie (graphe concurrent)

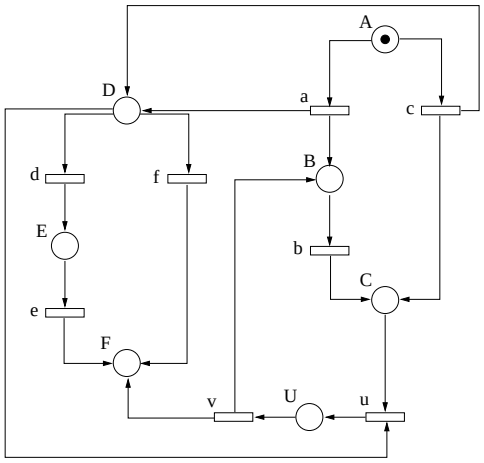


FIG. 3.8 – Deuxième contre exemple de détection de circuit

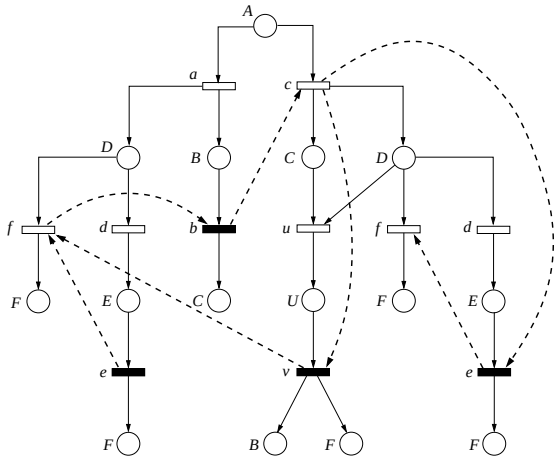


FIG. 3.9 – Détection erronée d'une séquence infinie (graphe simple)

- **Graphe concurrent** Un raccourci a comme unique successeur son image par ϕ et une image a comme successeurs tous les raccourcis causalement plus grands ou parallèles.
- **Graphe simple** Un raccourci a comme unique successeur son image par ϕ et une image a comme successeurs tous les raccourcis causalement plus grands.

En réduisant un comportement infini par les raccourcis rencontrés dans le préfixe, nous construisons un chemin dans le graphe concurrent. Alors qu'un chemin infini dans le graphe induit un comportement infini. Toutefois, aucun de ces deux graphes ne permet de décider de façon définitive de la présence d'un comportement infini. Le graphe concurrent peut contenir un circuit alors que le système ne peut pas réaliser de séquence infinie (voir le contre exemple présenté dans les figures 3.8 et 3.9). Le graphe simple peut ne pas faire apparaître de circuit alors que le système a la capacité de réaliser une séquence infinie (voir le contre exemple présenté dans les figures 3.6 et 3.7).

Une manière simple de contourner le problème est utilisée l'inclusion comme ordre adéquat (i.e. $C \prec C' \Rightarrow C \subset C'$). En effet, un réseau de Petri peut réaliser une séquence infinie si et seulement si le préfixe adéquat a au moins un raccourci.

Exemple 16 *Considèrons le préfixe fini $\{e_5, e_7, e_8\}$ défini à partir du processus arborescent de la figure 3.3. L'image de chacun des raccourcis est incluse dans sa configuration locale. Ainsi, à chaque raccourci correspond un comportement infini du réseau.*

Toutefois, cet ordre impose des conditions fortes sur les raccourcis et produit des préfixes de grande taille. Nous introduisons des conditions plus fines basées sur un couple d'ordres qui permettent la détection des comportements infinis.

Définition 3.30 (Couple adéquat d'ordres) *Un couple de relations $\langle \preceq_1, \prec_2 \rangle$ sur les configurations d'un dépliage (S, h) est un couple adéquat d'ordres si :*

- *$\preceq_1$ est un ordre adéquat,*
- *$\preceq_2$ est un pré-ordre (i;e. $\preceq_2$ est réflexive et transitive),*
- *$\prec_2$ raffine $\subset$ (i.e $C_1 \subset C_2 \Rightarrow (C_1 \preceq_2 C_2) \wedge (C_2 \not\preceq_2 C_1)$),*
- *$\preceq_1$ et $\preceq_2$ sont simultanément compatibles avec l'extension : pour toute transition t, pour toutes configurations C_1, C_2 de S vérifiant $h(Cut(C_1)) = h(Cut(C_2))$*

$$(C_1 \prec_1 C_2) \wedge (C_1 \succeq_2 C_2) \Rightarrow \\ \forall C_2' \in C_2 \cdot t, \exists C_1' \in C_1 \cdot t : (C_1' \prec_1 C_2') \wedge (C_1' \succeq_2 C_2')$$

La définition des préfixes adéquats doit être revue pour prendre en compte ce nouveau type d'ordre.

Définition 3.31 (Préfixe fini doublement adéquat) *Un préfixe fini Cutoff de $\langle S, h \rangle$ est doublement adéquat ssi il existe un couple adéquat d'ordres $\langle \preceq_1, \preceq_2 \rangle$ et une application $\phi : Cutoff \mapsto Conf(Cutoff)$ tels que $\forall e \in Cutoff$:*

- *$h(Cut(\phi(e))) = h(Cut([e]))$,*
- *$\phi(e) \prec_1 [e]$,*
- *$(\phi(e) \succ_2 [e]) \vee (C \subset [e])$.*

Les nouvelles contraintes imposées par le couple d'ordre, nous permettent d'avoir une caractérisation simple des comportements infinis représentés au sein d'un préfixe doublement adéquat.

Proposition 3.32 (Préfixe fini doublement adéquat) *Soit Cutoff un préfixe fini doublement adéquat de $\langle S, h \rangle$. R admet une séquence infinie à partir de m_0 ssi $\exists e \in Cutoff$ tel que $\phi(e) \subset [e]$. De plus, si $\langle R, m_0 \rangle$ est borné alors il existe au moins un préfixe fini doublement adéquat pour tout couple adéquat d'ordres.*

3.4 Graphes de préfixes finis

Nous avons vu jusqu'à présent en quoi les préfixes complets peuvent être un outil pour la vérification de propriétés de sûreté et la détection de circuits. Cependant, les techniques présentées ne permettent pas en l'état de décider de la satisfaction de formules de logique temporelle. En effet, les algorithmes de vérification nécessitent la détection de circuits particuliers, ce que ne nous permettent pas les préfixes doublement adéquats de la section 3.3.4. Dans cette section, nous présentons une structure particulière - des *graphes de préfixes finis* - adaptée à la vérification de formules de logique temporelle à temps linéaire.

3.4.1 Définition

Les graphes que nous étudions ont pour sommets des préfixes finis d'un réseau de Petri pour différentes valeurs de marquages initials. Les événements délimitant chaque préfixe sont maintenant partitionnés en deux sous-ensembles. Le premier est consitité de raccourcis proprement dit alors que les événements du second conduisent à des sommets successeurs et sont nommés des *ponts* (i.e. *Bridge*). Intuitivement, un raccourci indique que le franchissement de la transition correspondante conduit vers un état représenté dans le préfixe courant alors que pour un pont, son franchissement conduit vers l'état initial d'un préfixe successeur. L'objectif est de faire apparaître les circuits nécessaires à la vérification au niveau du graphe.

Pour tout marquage accessible m de $\langle R, m_0\rangle$, on note $\langle \mathcal{S}_m, h_m\rangle$ le dépliage du réseau $\langle R, m\rangle$.

Définition 3.33 (Graphe de préfixes finis) *Soit G un ensemble de quadruplets de la forme $\langle m, Cutoff, Bridge, \delta\rangle$ où m est un marquage de $Reach(R, m_0)$, $Cutoff \cup Bridge$ est un préfixe fini de $\langle \mathcal{S}_m, h_m\rangle$ et δ une application de $Bridge$ vers G. Pour tout g de G, nous notons $g = \langle m_g, Cutoff_g, Bridge_g, \delta_g\rangle$ et $h_g = h_{m_g}$. Le triplet $(G, \rightarrow, g_0)$ est un graphe de préfixes finis de $\langle R, m_0\rangle$ si*

- $g_0 \in G \wedge m_{g_0} = m_0$,
- $\forall g \in G, \forall e \in Bridge_g, m_{\delta_g(e)} = h_g(Cut([e]))$,
- $\forall g_1, g_2 \in G : g_1 \rightarrow g_2 \Rightarrow \exists e \in Bridge_{g_1}, g_2 = \delta_{g_1}(e)$,
- $\forall g \in G$, *il existe un chemin dans* $(G, \rightarrow, g_0)$ *de* g_0 *à* g.

Lorsque la transition correspondante à un pont est franchie, le marquage atteint est celui associé à la configuration locale de ce pont. Les deux premiers points de la définition nous assurent que les marquages initiaux, pour lesquels les différents préfixes sont définis, sont corrects (pour le sommet initial et pour chacun des ponts). Les deux derniers points de la définition imposent qu'aucun sommet ou arc inutile n'est pris en compte.

Exemple 17 *La figure 3.10 présente un graphe de préfixes finis du réseau de la figure 3.2. Les raccourcis sont représentés par des transitions noires et les ponts par des transitions grisées.*

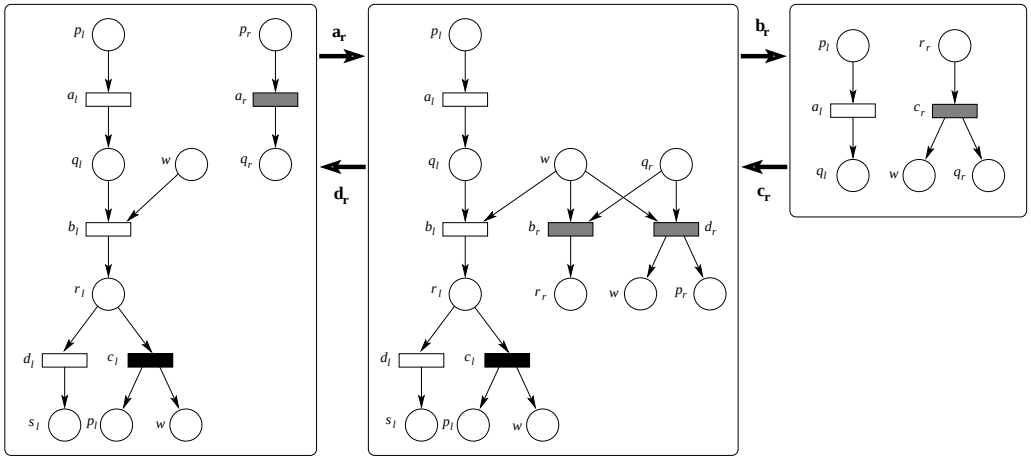


FIG. 3.10 – Un graphe de préfixes finis du réseau de la figure 3.2

Un graphe de préfixes finis est considéré comme étant complet si chaque état accessible est représenté dans au moins un sommet du graphe.

Définition 3.34 (Graphe complet de préfixes finis) *Soit $(G, \rightarrow, g_0)$ un graphe de préfixes finis de $\langle R, m_0\rangle$. Le graphe est complet si*

$$\bigcup_{g\in G} Reach(Cutoff_g \cup Bridge_g) = Reach(R, m_0)$$

Les ordres adéquats de la définition 3.27 doivent être adaptés à l'introduction des ponts. Ceci nous conduit à introduire la notion d'ordres fortement adéquats.

Définition 3.35 (Ordre fortement adéquat) *Soit $\preceq$ un ordre sur les configurations de tous les dépliages $\langle S_m, h_m\rangle$ de R tel que $m \in Reach(R, m_0)$. L'ordre $\preceq$ est fortement adéquat si :*

- *$\forall m \in Reach(R, m_0)$, $\preceq$ est adéquat sur les configurations de $\langle S_m, h_m\rangle$,*
- *$\preceq$ est compatible par rapport à la réduction de préfixe : $\forall m \in Reach(R, m_0), \forall C \in Conf(S_m), \forall e \in C$, soit $m' = h_m(Cut([e]))$,*

$$\exists C' \in Conf(S_{m'}) : (C' \prec C) \wedge (h_{m'}(C') = h_m(C \setminus [e]))$$

On peut noter que les ordres fortement adéquats portent sur les configurations d'une famille de dépliages et que la définition nous assure que lorsqu'un préfixe commun à deux configurations comparables est supprimé, la relation est préservée. Cette adaptation des ordres adéquats permet la définition de graphes de préfixes finis munis des propriétés de complétude.

Définition 3.36 (Graphe adéquat de préfixes finis) *Un graphe de préfixes finis $(G, \rightarrow, g_0)$ est adéquat ssi il existe un ordre fortement adéquat $\preceq$ et $\forall g \in G$, il existe une application $\phi_g : Cutoff_g \mapsto Conf(Cutoff_g \cup Bridge_g)$ satisfaisant :*

$$h_g(Cut(\phi_g(e))) = h_g(Cut([e])) \wedge \phi_g(e) \prec [e]$$

Proposition 3.37 (Graphe adéquat de préfixes finis) *Si un graphe de préfixes finis est adéquat alors il est un graphe complet.*

Il est clair que lorsque le réseau est borné, il existe au moins un graphe adéquat de préfixes finis pour tout ordre fortement adéquat. En effet, tout préfixe fini adéquat peut être vu comme un graphe adéquat composé d'un seul sommet.

A tout circuit du graphe correspond une classe d'équivalence de séquences infinies du réseau. Toutefois, des comportements infinis peuvent être représentés dans les sommets et ne pas apparaître au niveau du graphe. La détection de tels comportements infinis est primordial dans le cadre de la vérification. Une adaptation des couples d'ordres adéquats de la section 3.3.4 est proposée pour mettre en évidence ces comportements.

Définition 3.38 (Couple fortement adéquat d'ordres) *Soit $\langle \preceq_1, \preceq_2 \rangle$ un couple de relations sur les configurations de tous les dépliages $\langle S_m, h_m \rangle$ de R tel que $m \in Reach(R, m_0)$. Le couple $\langle \preceq_1, \preceq_2 \rangle$ est fortement adéquat si :*

- *$\preceq_1$ est fortement adéquat,*
- *$\forall m \in Reach(R, m_0)$, $\langle \preceq_1, \preceq_2 \rangle$ est un couple adéquat d'ordres sur les configurations de $\langle S_m, h_m \rangle$.*

Un graphe de préfixes finis construit sur un couple fortement adéquat d'ordres est dit doublement adéquat.

Définition 3.39 (Graphe de préfixes finis doublement adéquats) *Un graphe de préfixes finis $(G, \rightarrow, g_0)$ est doublement adéquat ssi il existe un couple fortement adéquat d'ordres $\langle \preceq_1, \preceq_2 \rangle$ et $\forall g \in G$, il existe une application ϕ_g : $Cutoff_g \mapsto Conf(Cutoff_g \cup Bridge_g)$ satisfaisant :*

- *$h_g(Cut(\phi_g(e))) = h_g(Cut([e]))$,*
- *$\phi_g(e) \prec_1 [e]$,*
- *$(\phi_g(e) \succ_2 [e]) \vee (\phi_g(e) \subset [e])$.*

Nous sommes maintenant en mesure de caractériser l'existence d'une séquence infinie du réseau dans le graphe de préfixes finis.

Proposition 3.40 (Graphe de préfixes finis doublement adéquats) *Soit $(G, \rightarrow, g_0)$ un graphe de préfixes finis doublement adéquats de $\langle R, m_0 \rangle$. $\langle R, m_0 \rangle$ admet une séquence infinie à partir de m_0 ssi une des conditions suivantes est vérifiée :*

- *Il existe un circuit dans le graphe $(G, \rightarrow, g_0)$,*
- *$\exists g \in G, \exists e \in Cutoff_g : \phi_g(e) \subset [e]$*

3.4.2 Logique linéaire événementielle

Dans cette section, nous montrons comment les graphes de préfixes finis doublement adéquats peuvent être employés pour la vérification de formules de logique temporelle à temps linéaire. Dans un premier temps, nous nous focalisons sur une version événementielle de LTL. La section suivante traite de

la logique propositionnelle. Il est important de noter que nous limitons notre étude aux réseaux de Petri bornés.

Dans la logique que nous considérons, seul un sous-ensemble des transitions du réseau est observé (i.e. cela correspond à un réseau étiqueté tel que l'étiquette de chaque transition est soit la transition elle même, et elle est alors observée, soit le mot vide). La prise en compte des séquences maximales, alors que les transitions peuvent être étiquetées par le mot vide, conduit à traiter les séquences divergentes (i.e. des séquences infinies dont un suffixe n'est composé que de franchissements de transitions étiquetées par le mot vide).

Le principe général de la méthode consiste à construire un système de transitions étiquetées équivalent au graphe des états accessibles du point de vue de la satisfaction de la formule considérée. Bien entendu, l'objectif est que ce système de transitions soit plus petit que le graphe des états accessibles.

Ce système de transitions étiquetées est construit à partir d'un graphe de préfixes finis dit compatible. Le but est de faire en sorte que toutes les occurrences des transitions observées apparaissent au niveau du graphe et non pas à l'intérieur d'un des préfixes.

Définition 3.41 (Graphe compatible de préfixes finis) *Soit $\mathcal{O}$ un sous-ensemble de transitions observables. Un graphe de préfixes finis $(G, \rightarrow, g_0)$ de $\langle R, m_0\rangle$ est compatible par rapport à $\mathcal{O}$ si $\forall g \in G, \forall e \in Event(Cutoff_g \cup Bridge_g) \cup Cutoff_g, h_g(e) \notin \mathcal{O}$.*

Deux types de comportements peuvent ne pas apparaître directement au niveau du graphe. En effet, un sommet non terminal du graphe peut très bien contenir un état bloquant. De même, des circuits non représentés au niveau du graphe peuvent être cachés au sein d'un préfixe. Il est à noter que les circuits de ce type correspondent à des séquences divergentes. Le graphe compatible de préfixes finis doit donc être complété pour mettre en évidence les comportements bloquants et divergents.

Le graphe complété est appelé un *graphe d'observation événementiel*. Le principe de construction est relativement simple. Un état artificiel (nommé $\perp$) et accessible par tous les préfixes contenant un état bloquant est ajouté. De plus, un arc bouclant est ajouté à chaque préfixe contenant un comportement infini. Enfin, tous les arcs sont étiquetés soit par une transition observée (s'ils correspondent à l'occurrence d'une telle transition) soit par le mot vide (dans le cas contraire et pour les arcs complémentaires). La détection des états bloquants est réalisée par les algorithmes [61, 45] décrits dans la section 3.3.3. La détection des suffixes divergents est obtenue par l'emploi d'un couple fortement adéquat d'ordres.

Définition 3.42 (Graphe d'observation événementiel) *Soit $\mathcal{O}$ un sous-ensemble de transitions observées. Soit $(G, \rightarrow, g_0)$ un graphe de préfixes finis doublement adéquats et compatible par rapport à $\mathcal{O}$. Le graphe d'observation événementiel est défini par $(\mathcal{O} \cup \{\tau\}, G \cup \{\perp\}, \longrightarrow_{\mathcal{O}}, g_0)$, où la relation de transition est donnée par : $\forall g_1, g_2 \in G, \forall t \in \mathcal{O}$,*

- $g_1 \xrightarrow{t}_{\mathcal{O}} g_2$ *ssi* $\exists e \in Bridge_{g_1} : h_{g_1}(e) = t \wedge \delta_{g_1}(e) = g_2$

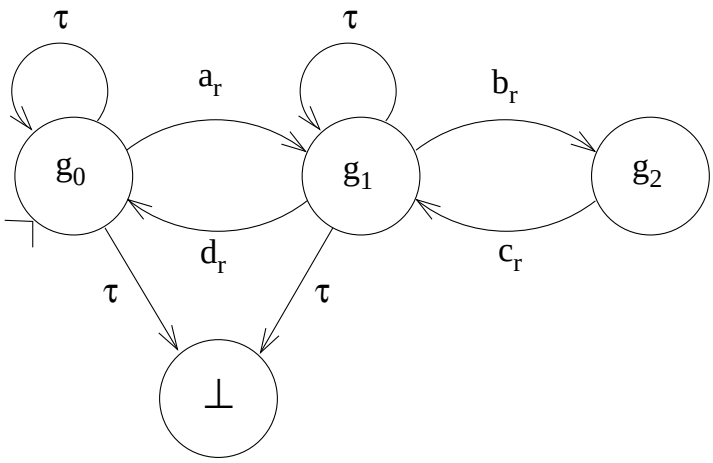


FIG. 3.11 – Le graphe d'observation événementiel correspondant

- $g_1 \xrightarrow{\tau}_{\mathcal{O}} g_2$ *ssi* $\begin{cases} & \big(\exists e \in Bridge_{g_1} : (h_{g_1}(e) \notin \mathcal{O}) \wedge (\delta_{g_1}(e) = g_2)\big) \\ \vee & \big((\exists e \in Cutoff_{g_1} : \phi_{g_1}(e) \subset [e]) \wedge (g_1 = g_2)\big) \end{cases}$
- $g_1 \xrightarrow{\tau}_{\mathcal{O}} \perp$ *ssi* $\begin{cases} \exists C \in Conf(Cutoff_{g_1} \cup Bridge_{g_1}) : \\ \quad \forall e \in Cutoff_{g_1} \cup Bridge_{g_1}, \exists e' \in C, e \sharp e' \end{cases}$
- $\neg(\perp \xrightarrow{t}_{\mathcal{O}} g_2) \wedge \neg(\perp \xrightarrow{\tau}_{\mathcal{O}} g_2) \wedge \neg(g_1 \xrightarrow{t}_{\mathcal{O}} \perp)$

Exemple 18 *Le graphe de préfixes finis de la figure 3.10 est un graphe doublement adéquat du réseau de la figure 3.2. De plus, il est compatible avec l'ensemble de transitions* $\{a_r, b_r, c_r, d_r\}$. *Le graphe d'observation événementiel correspondant est donné dans la figure 3.11. Les nœuds* g_0 *et* g_1 *contiennent une représentation de l'état bloquant* $s_l + q_r$ *et ont donc comme successeur l'état artificiel* $\perp$. *De même, ils ont des raccourcis ayant des images causalement plus petites. En conséquence, des arcs bouclants étiquetés par la transition muette* τ *leur sont associés.*

Pour montrer que le graphe d'accessibilité et le graphe d'observation sont équivalents du point de vue de la satisfaction d'une formule, nous définissons la projection d'un mot (fini ou non) $\sigma \in T^\omega$ sur un sous-ensemble de transitions $\mathcal{O} \subseteq T$ noté $\sigma_{\lfloor \mathcal{O}}$. La projection de σ sur $\mathcal{O}$ est définie par induction sur sa longueur :

- $\lambda_{\lfloor \mathcal{O}} = \lambda$,
- $(t \cdot \sigma)_{\lfloor \mathcal{O}} = \begin{cases} t \cdot \sigma_{\lfloor \mathcal{O}} & \text{si } t \in \mathcal{O} \\ \sigma_{\lfloor \mathcal{O}} & \text{si } t \notin \mathcal{O} \end{cases}$

On peut remarquer que la projection d'une séquence infinie n'est pas nécessairement infinie (si un suffixe de la séquence est composé uniquement de transitions n'appartenant pas à $\mathcal{O}$ alors la projection de la séquence est finie).

Nous notons $L_{\lfloor \mathcal{O}}(R, m_0)$ l'ensemble composé de la projection des séquences finies de franchissements de $\langle R, m_0 \rangle$ et $L_{Max \lfloor \mathcal{O}}(R, m_0)$ la projection de ses séquences maximales (séquences infinies de franchissements ou conduisant à un

état bloquant). Pour un graphe d'observation événementiel $\mathcal{G}$, les notations similaires $L_{\lfloor\mathcal{O}}(\mathcal{G})$ et $L_{Max\lfloor\mathcal{O}}(\mathcal{G})$ sont employées pour désigner la projection de ses mots finis et maximaux.

Proposition 3.43 (Graphe d'observation événementiel) *Soit $\mathcal{O}$ un sous-ensemble de transitions observées. Soit $\mathcal{G}$ un graphe d'observation événementiel d'un graphe de préfixes finis doublement adéquats et compatible par rapport à $\mathcal{O}$.*

- $L_{\lfloor\mathcal{O}}(R, m_0) = L_{\lfloor\mathcal{O}}(\mathcal{G})$
- $L_{Max\lfloor\mathcal{O}}(R, m_0) = L_{Max\lfloor\mathcal{O}}(\mathcal{G})$

Il est important de noter que cette proposition nous assure que le graphe d'observation événementiel peut être le support de la vérification de toute formule de logique temporelle à temps linéaire portant sur un sous-ensemble des événements du système.

Exemple 19 *On peut remarquer que le graphe d'observation événementiel de la figure 3.11 peut être le support de la vérification d'une formule de logique temporelle telle que $\mathbf{GF}(\mathbf{X}_{\{a_r\}} \Rightarrow \mathbf{FX}_{\{c_r\}})$. La taille de ce graphe est de 4 nœuds et 8 arcs alors que celle du graphe d'accessibilité du réseau de la figure 3.2 est de 10 nœuds et 19 arcs.*

3.4.3 Logique linéaire propositionnelle

Un travail similaire peut être fait dans le cadre d'une logique propositionnelle. L'équivalence recherchée ici est connue dans la littérature sous le nom d'équivalence bégayante et il a été montré que cette relation d'équivalence est idoïne pour la vérification de formule LTL privée de l'opérateur X. Dans le contexte de cette logique, il n'est pas nécessaire de distinguer dans une séquence deux états successifs vérifiant le même sous-ensemble de propositions atomiques. Ceci nous permet de définir l'ensemble des transitions devant être observées.

Définition 3.44 (Transitions observées) *Soit AP un ensemble de propositions atomiques. Soit $v : Reach(R, m_0) \mapsto 2^{AP}$ une fonction de valuation. Un ensemble de transitions $\mathcal{O}$ est observé si $\forall t \notin \mathcal{O}, \forall m_1, m_2 \in Reach(R, m_0)$,*

$$m_1[t\rangle m_2 \Rightarrow v(m_1) = v(m_2)$$

Ici encore, le graphe compatible de préfixes finis doit être complété pour faire apparaître les séquences bloquantes et les suffixes divergents. Tous les arcs sont maintenant étiquetés par l'ensembles des propositions atomiques satisfaites par les états représentés dans le préfixe.

Définition 3.45 (Graphe d'observation propositionnel) *Soit AP un ensemble de propositions atomiques. Soit $v : Reach(R, m_0) \mapsto 2^{AP}$ une fonction de valuation. Soit $\mathcal{O}$ un sous-ensemble de transitions observées pour AP. Soit $(G, \rightarrow, g_0)$ un graphe de préfixes finis doublement adéquats et compatible par*

rapport à $\mathcal{O}$. Le graphe d'observation propositionnel est défini par $(2^{AP}, G \cup \{\bot\}, \longrightarrow_{\mathcal{O}}, g_0)$, où la relation de transition est donnée par : $\forall g_1, g_2 \in G, \forall a \in 2^{AP}$,

- $g_1 \xrightarrow{a}_{\mathcal{O}} g_2$ *ssi* $a = v(m_{g_1}) \wedge \left\{ \begin{array}{ll} & (g_1 \rightarrow g_2) \\ \vee & \big((\exists e \in \mathit{Cutoff}_{g_1}, \phi_{g_1}(e) \subset [e]) \wedge (g_1 = g_2)\big) \end{array} \right.$
- $g_1 \xrightarrow{a}_{\mathcal{O}} \bot$ *ssi* $a = v(m_{g_1}) \wedge \left\{ \begin{array}{l} \exists C \in \mathit{Conf}(\mathit{Cutoff}_{g_1} \cup \mathit{Bridge}_{g_1}) : \\ \quad \forall e \in \mathit{Cutoff}_{g_1} \cup \mathit{Bridge}_{g_1}, \exists e' \in C, e \sharp e' \end{array} \right.$
- $\neg(\bot \xrightarrow{a}_{\mathcal{O}} g_2) \wedge \neg(g_1 \xrightarrow{a}_{\mathcal{O}} \bot)$

Les langages correspondant au réseau de Petri et au graphe d'observation doivent être redéfinis par rapport à la fonction d'étiquetage v et à la notion de bégaiement. Le modèle considéré est maintenant une séquence (finie ou infinie) sur l'alphabet 2^{AP}. Soit $\sigma = x_1 \cdot x_2 \cdots$ un mot fini ou infini sur 2^{AP}. On note σ^k le suffixe de σ commençant par x_k et $|\sigma|$ la longueur de σ. L'extraction de σ par rapport à v, notée $\sigma_{\lfloor v}$ est définie par :

$$\begin{array}{rcl} \sigma_{\lfloor v} & = & \text{si } |\sigma| \leq 1 \text{ alors } \sigma \\ & & \text{sinon si } |\sigma| = \infty \wedge \forall i > 1, x_1 = x_i \text{ alors } \sigma \\ & & \text{sinon si } x_1 = x_2 \text{ alors } (\sigma^2)_{\lfloor v} \\ & & \text{sinon } x_1 \cdot (\sigma^2)_{\lfloor v} \end{array}$$

Nous notons $L_{\lfloor v}(R, m_0)$ l'ensemble composé de l'extraction des séquences finies d'états de $\langle R, m_0 \rangle$ et $L_{Max \lfloor \mathcal{O}}(R, m_0)$ l'extraction de ses séquences maximales (séquences infinies d'états ou conduisant à un état bloquant). Pour un graphe d'observation propositionnel $\mathcal{G}$, les notations similaires $L_{\lfloor v}(\mathcal{G})$ et $L_{Max \lfloor v}(\mathcal{G})$ sont employées pour désigner l'extraction de ses mots finis et maximaux.

Proposition 3.46 (Graphe d'observation propositionnel) *Soit AP un ensemble de propositions atomiques. Soit $v : \mathit{Reach}(R, m_0) \mapsto 2^{AP}$ une fonction de valuation. Soit $\mathcal{O}$ un sous-ensemble de transitions observées pour AP. Soit $\mathcal{G}$ un graphe étiqueté d'observation d'un graphe de préfixes finis doublement adéquats et compatible par rapport à $\mathcal{O}$.*

- $L_{\lfloor v}(R, m_0) = L_{\lfloor v}(\mathcal{G})$
- $L_{Max \lfloor v}(R, m_0) = L_{Max \lfloor v}(\mathcal{G})$

3.5 Dépliages de réseaux de Petri symétriques

Dans cette section, nous introduisons les réseaux de Petri symétriques et montrons comment nous pouvons tenir compte des symétries pour réduire la taille du préfixe complet. Notre objectif principal est l'étude de système communicant par des files. Nous comparons plusieurs modélisations de files et analysons les conséquences du choix d'un modèle sur la construction d'un préfixe complet.

3.5.1 Réseaux de Petri symétriques

Les symétries sont définies comme un groupe d'automorphismes sur les éléments du réseaux.

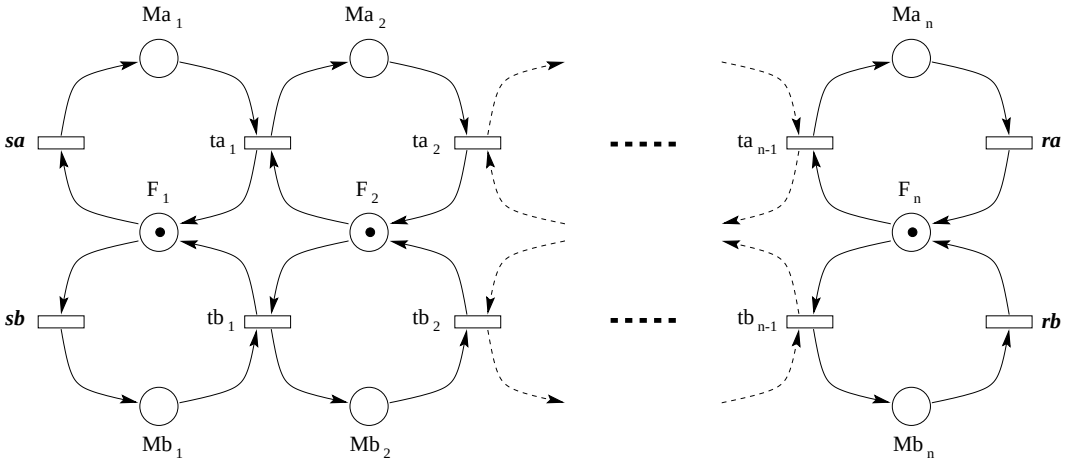


FIG. 3.12 – Un premier modèle d'une file bornée

Définition 3.47 *Un réseau de Petri symétrique est un couple* $\langle R, \mathcal{S} \rangle$ *où :*

- $R = \langle P, T, Pre, Post, \Sigma, \lambda \rangle$ *est réseau étiqueté,*
- $\mathcal{S}$ *est un groupe d'automorphismes de* R *(i.e. homomorphismes bijectifs de* R *dans* R*) invariants par rapport à la fonction d'étiquetage* λ *:* $\forall t \in T, \forall g \in \mathcal{S} : \lambda(t) = \lambda(h(t))$.

Un réseau symétrique marqué est une structure $\langle R, \mathcal{S}, m_0 \rangle$ *où* m_0 *est un marquage de* R.

A partir de cette définition, nous introduisons la définition de marquages équivalents.

Définition 3.48 *Soit* $\langle R, \mathcal{S} \rangle$ *un réseau symétrique. Deux marquages* m *et* m' *de* R *sont équivalents (noté* $m \equiv m'$*) si* $\exists g \in \mathcal{S}$ *tel que* $g_l(m) = m'$*. Pour un marquage* m *de* R*, nous notons* $\widehat{m}$ *l'ensemble des marquages équivalents à* m*. Si* M *est un ensemble de marquage, nous notons* $\widehat{M}$ *l'ensemble* $\{\widehat{m} \mid \exists m' \in M : m \equiv m'\}$.

L'introduction de files dans les réseaux de Petri conduit à une extension stricte des réseaux de Petri fini. Toutefois, quand la taille des files est connue à priori, il est possible de modéliser une file par un réseau de Petri. Généralement, dès que la taille de la file n'est pas connue ou infinie, la modélisation conduit à un réseau de Petri infini. Nous allons étudier ces deux cas.

La figure 3.12 donne une première modélisation d'une file bornée. Ici, la file peut recevoir deux type de messages (a et b). Les transitions sx correspondent à l'ajout d'un message x, et les transitions rx au retrait d'un message x. Chaque position $i \in [1, n[$ de la file est modélisée par trois places $\{Ma_i, Mb_i, F_i\}$ où les deux premières places indiquent la présence d'un message et la dernière l'absence de message. Evidemment, parmi ces places, une seule est marquée. Quand un message est ajouté, il est rangé dans la première position. Puis, le message passe à travers toutes les positions avant d'être retiré. Le défaut majeur de cette modélisation est qu'un état de la file est généralement représenté par

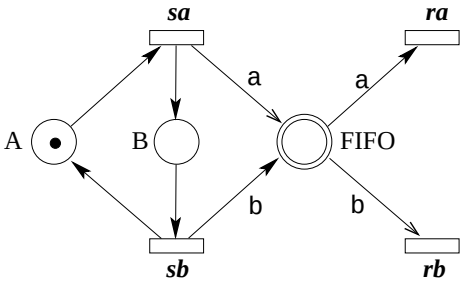


FIG. 3.13 – Un exemple de réseau à file

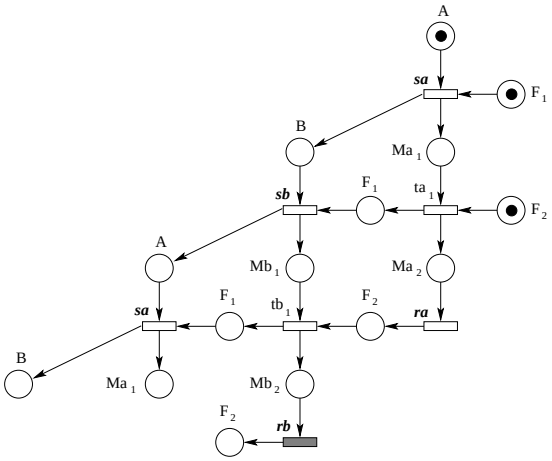


FIG. 3.14 – Préfixe complet du réseau de la figure 3.13 basé sur le premier modèle d'une file de taille 2

plusieurs marquages. Par exemple, si la file contient un message, ce message peut se trouver dans le réseau de Petri à n'importe quelle position. Ainsi, le modèle introduit de nombreux marquages et transitions intermédiaires.

Un préfixe complet du réseau de la figure 3.13 basé sur cette modélisation d'une file est donné figure 3.14. En généralisant le dépliage à celui d'une file de taille n, nous pouvons montrer qu'il contient $\frac{1}{2}(n+1)(n+2)+1$ événements alors que le système n'a que $2(n+1)$ états.

La deuxième modélisation d'une file est donnée figure 3.15. Elle est basée sur l'utilisation d'un tableau circulaire et de deux compteurs In et Out. La valeur de In (resp. Out) indique la position dans le tableau où un nouveau message doit être ajouté (resp. retiré). La figure 3.16 donne le préfixe fini du réseau de la figure 3.13. En généralisant le dépliage à une file de taille n, nous pouvons montrer qu'il croît linéairement avec la taille de la file. Malgré tout, notre nouvelle modélisation introduit des duplications des états de la file. En effet, deux états identiques de la file peuvent avoir des valeurs de compteurs In et Out différentes. Cependant ce problème peut être résolu en introduisant des symétries dans le modèle et en adaptant les règles de coupures utilisées dans le calcul du préfixe fini. Pour celà, nous devons étendre les technique de dépliage aux réseaux symétriques.

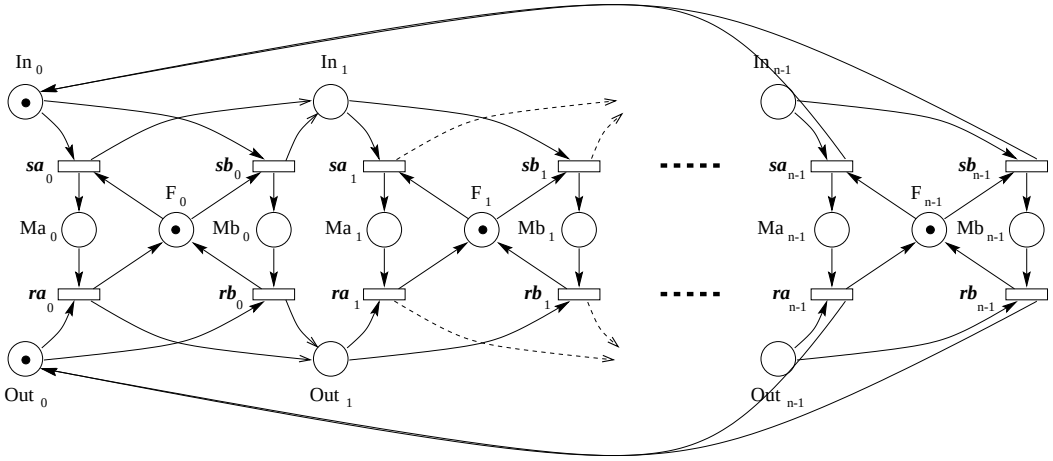


FIG. 3.15 – La deuxième modélisation d'une file bornée

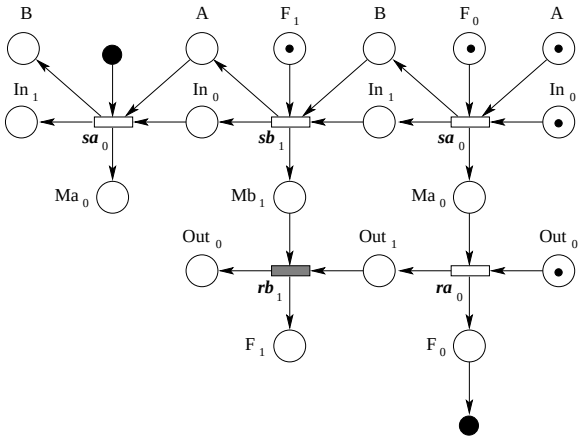


FIG. 3.16 – Un préfixe complet du réseau de la figure 3.13 basé sur le deuxième modèle d'une file de taille 2

3.5.2 Préfixe fini complet d'un réseau symétrique

Un processus arborescent d'un réseau symétrique est simplement un processus du réseau de Petri sous-jacent.

Définition 3.49 *Un processus arborescent d'un réseau symétrique $\langle R, \mathcal{S}, m_0\rangle$ est un processus arborescent du réseau $\langle R, m_0\rangle$.*

Les symétries permettent de relaxer la condition de complétude d'un préfixe.

Définition 3.50 *Un préfixe fini Cutoff d'un dépliage $\langle S, h\rangle$ d'un réseau symétrique $\langle R, \mathcal{S}, m_0\rangle$ est complet si $h(\widehat{Reach(Cutoff)}) = \widehat{Reach(R, m_0)}$.*

De manière analogue, nous redéfinissons la notion d'ordre adéquat et de préfixe adéquat.

Définition 3.51 (Ordre adéquat) *Soit $\langle S, h\rangle$ le dépliage d'un réseau symétrique $\langle R, \mathcal{S}, m_0\rangle$. Un ordre partiel $\preceq$ sur les configurations du dépliage est adéquat si*

- *$\preceq$ rafine $\sqsubseteq$,*
- *$\preceq$ est bien fondé,*
- *$\preceq$ est compatible avec l'extension : pour toute transition t et toute paire de configurations C_1, C_2 de S telles que $\exists g \in \mathcal{S}, g(h(Cut(C_1))) = h(Cut(C_2))$:*

$$C_1 \prec C_2 \Rightarrow \forall C_2' \in C_2 \cdot t, \exists C_1' \in C_1 \cdot g(t) : C_1' \prec C_2'$$

Définition 3.52 (Préfixe fini adéquat) *Un préfixe fini Cutoff de $\langle S, h\rangle$ est adéquat s'il existe un ordre adéquat $\preceq$ et une application $\phi : Cutoff \mapsto Conf(Cutoff)$ tels que $\forall e \in Cutoff : h(\widehat{Cut(\phi(e))}) = h(\widehat{Cut([e])}) \wedge (\phi(e) \prec [e])$.*

Proposition 3.53 (Préfixe fini adéquat) *Si un préfixe fini est adéquat alors il est complet.*

Nous pouvons noter que l'ordre de McMillan est adéquat. En effet, il ne prend pas en compte l'identité de l'événement.

La figure 3.17 présente le préfixe complet du réseau de la figure 3.13 utilisant une file de taille 5 (figure 3.15). Le groupe d'automorphisme est donné par $\mathcal{S} = \{g_k : P \cup T \to P \cup T \mid \forall v_i \in FIFO, g_k(v_i) = v_{i\oplus k} \wedge \forall v \notin FIFO, g_k(v) = v\}_{k\in[1,n]}$ où $\oplus$ représente l'opérateur modulo. Les symétries nous permettent de détecter au plus tôt la coupure sur l'événement rb. En effet, le marquage atteint après le franchissement de $[rb]$ est équivalent à l'état initial.

La modélisation d'une file non bornée est similaire à celle d'une file bornée : elle utilise simplement un tableau non borné plutôt qu'un tableau circulaire. Le modèle de la file est donné figure 3.18 et permet l'analyse de système à nombre fini d'états dont les bornes des files ne sont pas connues à priori.

La figure 3.19 donne un exemple de système borné utilisant une file non bornée, ainsi que son préfixe fini. Nous pouvons noter que l'analyse du préfixe nous permet de déduire la borne de la file (la borne est 3 sur l'exemple).

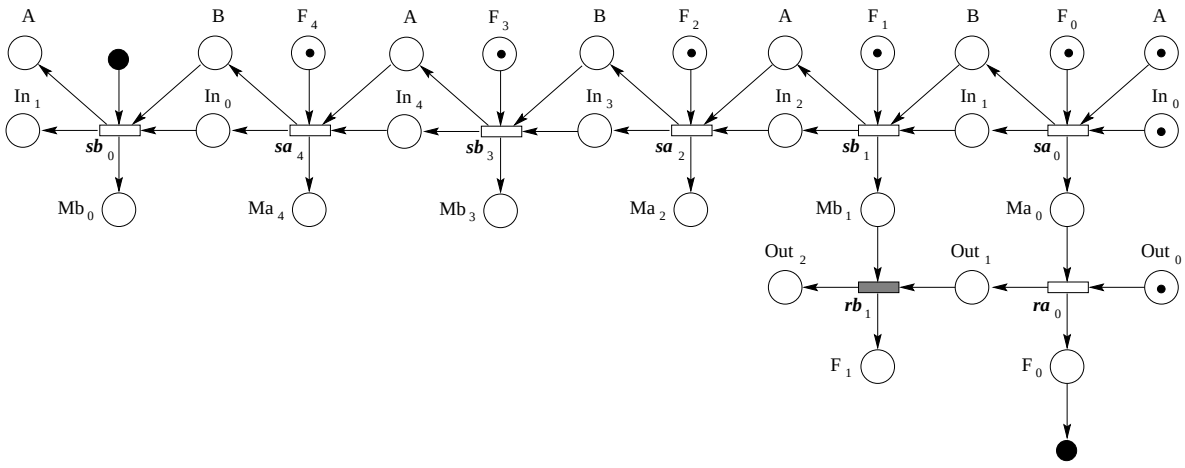


FIG. 3.17 – Un préfixe complet du réseau de la figure 3.13 basé sur le deuxième modèle d'une file de taille 5

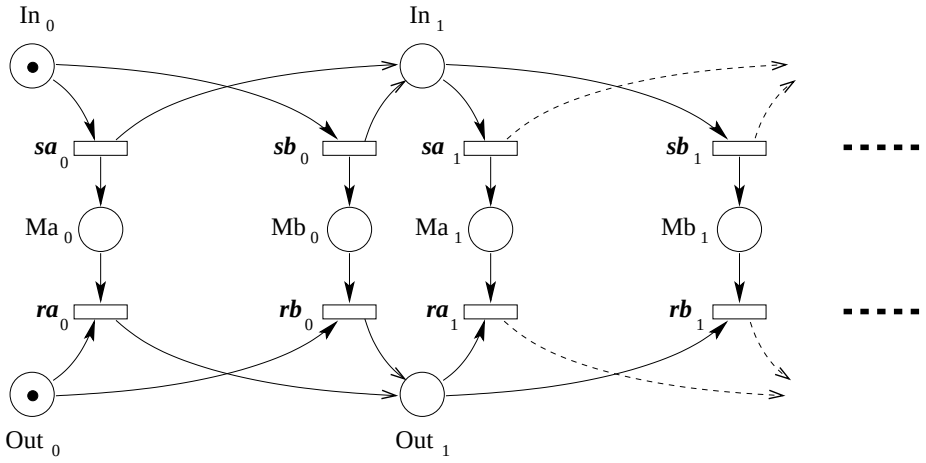


FIG. 3.18 – Modélisation d'une file non bornée

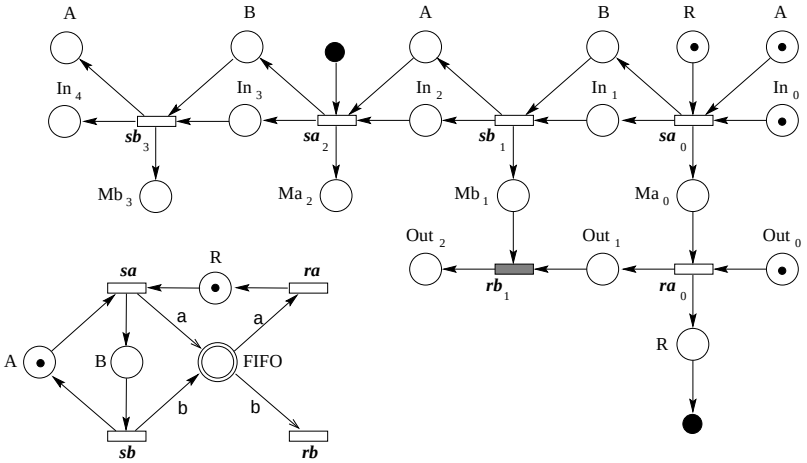


FIG. 3.19 – Un réseau à file non bornée et son préfixe complet

3.6 Dépliage d'un produit de réseaux symétriques

Considérer un système comme un ensemble de composantes interagissant est le point de nombreuses optimisations de méthodes de vérification. Dans cette partie, nous établissons les éléments théoriques aboutissant à la conception d'un algorithme de construction d'un préfixe complet basé sur l'analyse des dépliages des composants.

3.6.1 Produit de réseaux de Petri

Notre modèle est un produit de réseaux de Petri synchronisés sur des noms d'actions.

Définition 3.54 *Soit $R_1, \ldots, R_n$ des réseaux de Petri étiquetés. Notons $R_i = \langle P_i, T_i, Pre_i, Post_i, \Sigma_i, \lambda_i \rangle$ (nous supposons par commodité que les ensembles de places et les ensembles de transitions sont deux à deux disjoints). Le produit $R = \langle P, T, Pre, Post, \Sigma, \lambda \rangle$ des réseaux R_i est un réseau étiqueté où :*

- $P = \bigcup_i P_i$,
- $T = \{t \in \prod_i (T_i \cup \{\epsilon\}) \mid \exists a \in \cup_i \Sigma_i, \forall i : (a \in \Sigma_i \wedge t[i] \in T_i \wedge \lambda_i(t[i]) = a) \vee (a \notin \Sigma_i \wedge t[i] = \epsilon)\}$,
- $\forall t \in T, \forall i, \forall p \in P_i : Pre(t)(p) = Pre_i(t[i])(p)$ *et* $Post(t)(p) = Post_i(t[i])(p)$,
- $\Sigma = \bigcup_i \Sigma_i$,
- $\forall t \in T, \forall i \in [1, n], t[i] \neq \epsilon \Rightarrow \lambda(t) = \lambda_i(t[i])$.

Si les R_i ont m_i comme marquage initial alors R a $m_0 = \sum_i m_i$ comme marquage initial. Nous noterons $\bigotimes_i R_i$ (resp. $\bigotimes_i \langle R_i, m_i \rangle$) le produit des réseaux R_i (resp. des réseaux marqués $\langle R_i, m_i \rangle$).

Une propriété fondamentale pour l'étude du dépliage d'un produit est la notion de non-réentrance.

Définition 3.55 *Soit $R = \langle P, T, Pre, Post, \Sigma, \lambda, m_0 \rangle$ un réseau étiqueté marqué. Une étiquette a de Σ est non-réentrante sur R si $\forall m \in Reach(R, m_0), \forall t, t' \in T, \lambda(t) = \lambda(t') = a \Rightarrow Pre(t) + Pre(t') \not\subseteq m$. Soit $R = \bigotimes_i \langle R_i, m_i \rangle$ un produit de réseaux R_i. Le produit R est non-réentrant si $\forall a \in \Sigma, \forall i, j, (i \neq j \wedge a \in \Sigma_i \cap \Sigma_j) \Rightarrow a$ est non-réentrant sur R_i. Un tel produit sera appelé non-réentrant.*

Exemple 20 *La figure 3.20 présente deux réseaux R_1, R_2 et leur produit. Nous pouvons noter que le produit est non-réentrant. Par contre, si nous considérons les lettres x et y identiques, alors le produit perd sa propriété de non-réentrance.*

3.6.2 Dépliage d'un produit de réseaux de Petri

Pour la suite de notre étude, nous considérons un ensemble de réseaux étiquetés $\langle R_i, m_i \rangle$ et leur réseau produit $\langle R, m_0 \rangle = \bigotimes_i \langle R_i, m_i \rangle$. Nous noterons $\beta_i = \langle \langle B_i, E_i, In_i, Out_i \rangle, h_i \rangle$ le dépliage de $\langle R_i, m_i \rangle$ et $\beta = \langle \langle B, E, In, Out \rangle, h \rangle$ le dépliage du produit. En considérant les dépliages β_i comme des réseaux étiquetés, le produit $\bigotimes_i \beta_i$ est clairement défini. Noter que ce produit n'est pas

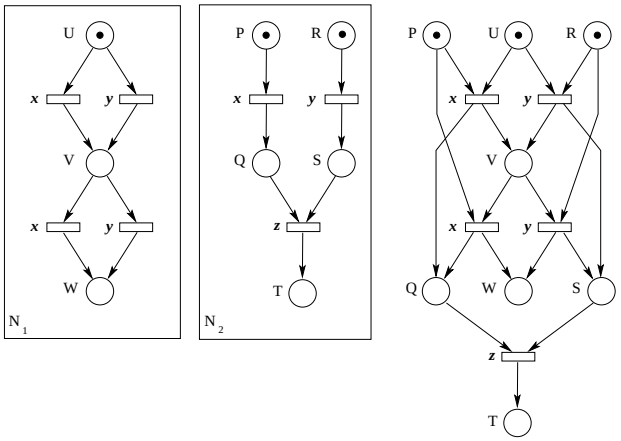


FIG. 3.20 – Un produit non-réentrant de réseaux

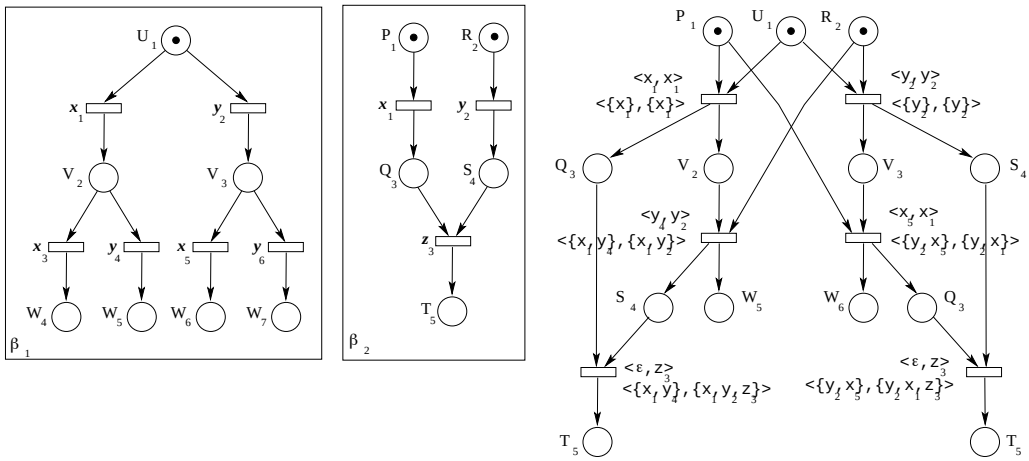


FIG. 3.21 – Processus arborescent d'un produit non-réentrant de réseaux de Petri

nécessairement un processus arborescent. Cependant, il sera utilisé comme une structure intermédiaire pour relier les éléments du dépliage β aux dépliages des composants β_i.

Proposition 3.56 *Il existe un homomorphisme* Ψ *de* β *dans* $\bigotimes_i \beta_i$ *tel que :*
- $\forall b \in B, \forall i : \Psi(b) \in B_i \Rightarrow h(b) = h_i(\Psi(b))$,
- $\forall e \in E, \forall i :$ *si* $\Psi(e)[i] \in E_i$ *alors* $h(e)[i] = h_i(\Psi(e)[i])$ *sinon* $h(e)[i] = \epsilon$.

Exemple 21 *La figure 3.21 donne un exemple d'homomorphisme* Ψ *du produit de réseaux de la figure 3.20. A gauche, nous avons les dépliages des composants ; à droite le dépliage du produit avec pour chaque élément, la valeur de la fonction* Ψ.

La fonction Ψ est utilisée pour caractériser le comportement de chaque composante à partir du comportement du produit. La proposition suivante établit

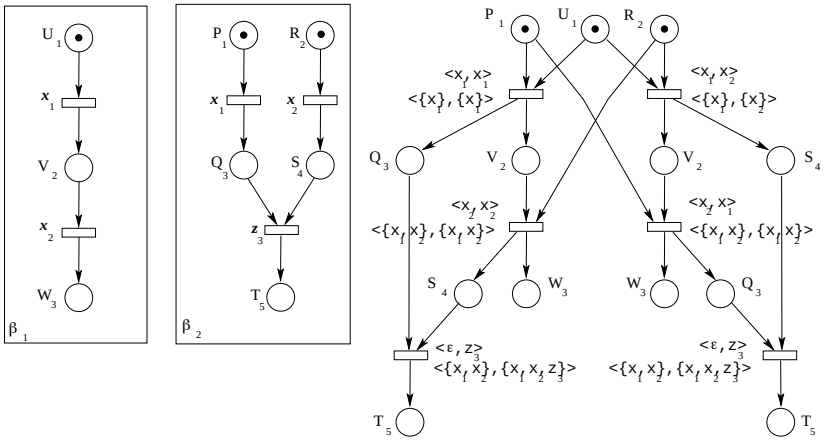


FIG. 3.22 – Processus arborescent d'un produit réentrant de réseaux de Petri

que la propriété de non-réentrance garantit que la fonction Φ est injective, et donc que l'image d'un élément du dépliage (une condition ou un événement) peut être utilisée pour identifier l'élément.

Proposition 3.57 *Soit C une configuration de β. Soit pour tout i, $C[i] = \{\Psi(e)[i] \mid e \in C \wedge \Psi(e)[i] \neq \epsilon\}$. L'ensemble $C[i]$ est une configuration de β_i et $\Psi(Cut(C)) = \bigcup_i Cut(C[i])$. De plus, si $\langle R, m_0\rangle$ est non-réentrant alors la fonction qui associe à une configuration C le tuple de configuration $\langle C[1], \ldots, C[n]\rangle$ est injective.*

La figure 3.22 donne un exemple de système réentrant pour lequel une configuration n'est pas identifiée de manière unique par sa projection sur les dépliages de ces composants. Les deux événements étiquetés $\langle \epsilon, z_3\rangle$ ont des configurations locales différentes donnant des projections identiques sur les composantes.

Une fonction importante pour la construction d'un préfixe fini est de déterminer si deux conditions b_1 et b_2 sont concurrentes. Traditionellement, ce test consiste à vérifier que l'union des deux configurations locales aux deux conditions forme une configuration. La proposition suivante conduit à la mise en œuvre d'une méthode modulaire pour réaliser ce test.

Proposition 3.58 *Soit C, C' deux configurations de β. Si $\langle R, m_0\rangle$ est non-réentrant, alors $C \cup C'$ est une configuration de β ssi $\forall i, C[i] \cup C'[i]$ est une configuration de β_i.*

3.6.3 Préfixe fini complet d'un produit de réseaux de Petri

Dans cette section, nous montrons comment combiner des ordres adéquats sur les configurations des dépliages des composants pour obtenir un ordre adéquat sur le dépliage du produit. Cette étude nous conduit à la notion de pré-ordre bien adéquat.

Définition 3.59 (Pré-ordre bien adéquat) *Un pré-ordre $\preceq$ sur les configurations d'un dépliage est bien adéquat si*

- $\preceq$ *rafine* $\sqsubseteq$,
- $\preceq$ *est bien fondé,*
- $\preceq$ *est compatible avec l'extension : pour toute transition* t *et toute paire de configurations* C_1, C_2 *de* S *telle que* $h(Cut(C_1)) = h(Cut(C_2))$ *:*
  1. $C_1 \prec C_2 \Rightarrow \forall C_2' \in C_2 \cdot t, \exists C_1' \in C_1 \cdot t : C_1' \prec C_2'$
  2. $C_1 \equiv_\preceq C_2 \Rightarrow \forall C_2' \in C_2 \cdot t, \forall C_1' \in C_1 \cdot t : C_1' \equiv_\preceq C_2'$

Notons que si $\preceq$ est un pré-ordre bien adéquat alors $\prec$ induit un ordre partiel adéquat. Cette nouvelle notion permet de combiner facilement des ordres adéquats pour créer de nouveaux ordres plus fins.

Proposition 3.60 *Soit* $\preceq_1, \preceq_2$ *deux pré-ordres bien adéquats sur les configurations d'un dépliage. Soit* $lex(\preceq_1, \preceq_2)$ *la relation définie par :*

$$C \ lex(\preceq_1, \preceq_2) \ C' \text{ ssi } C \preceq_1 C' \vee (C \equiv_{\preceq_1} C' \wedge C \preceq_2 C')$$

Alors la relation $lex(\preceq_1, \preceq_2)$ *est un pré-ordre bien adéquat.*

Un pré-ordre bien adéquat sur les configurations du dépliage définit un pré-ordre sur les configurations du dépliage du produit. En combinant lexicographiquement ces pré-ordres, il est possible de concevoir des ordres adéquats pour la construction modulaire du préfixe d'un produit. La proposition suivante précise les résultats de cette méthode.

Proposition 3.61 *Soit pour tout* i,$\preceq_i$ *un pré-ordre bien adéquat sur les configurations du dépliage du réseau* R_i*. Soit la relation* $\preceq_{[i]}$ *sur les configurations du dépliage du produit* R *des* R_i *définies par* $C \prec_{[i]} C'$ *si* $C[i] \prec_i C'[i]$*). Alors ces relations sont des pré-ordres bien adéquats. De plus, si* R *est non-réentrant et si pour tout* i, $\preceq_i$ *est un ordre total alors* $lex(\prec_{[1]}, \ldots, \prec_{[n]})$ *est un ordre total.*

La proposition précédente a des conséquences importantes. Elle permet de sélectionner le meilleur ordre en fonction du type de composantes. Par exemple, dans [54, 31], les composantes sont des machines à états, l'ordre choisi est total et peut être facilement testé. Dans la section suivante, nous généraliserons ce résultat à des produits de réseaux symétriques afin d'obtenir des ordres totaux adéquats pour des modèles manipulant des files et des machines à états.

3.6.4 Préfixe fini complet d'un produit de réseaux symétriques

Cette partie présente les adaptations attendues des résultats précédants pour des produits de réseaux symétriques. Nous donnons dans un premier temps la définition d'un produit de réseaux symétriques, puis redéfinissons la notion de pré-ordre bien adéquat et concluons par la proposition donnant un ordre adéquat bien adéquat à la construction modulaire.

Définition 3.62 *Soit* $\langle R_1, \mathcal{S}_1\rangle, \ldots, \langle R_n, \mathcal{S}_n\rangle$ *une famille de réseaux symétriques. Le produit* $\langle R, \mathcal{S}\rangle = \bigotimes_i \langle R_i, \mathcal{S}_i\rangle$ *est donné par* $\langle \bigotimes_i R_i, \prod_i \mathcal{S}_i\rangle$*. Un produit de réseaux symétriques marqués* $\langle \bigotimes_i \langle R_i, \mathcal{S}_i, m_i\rangle$ *est non-réentrant si le produit* $\bigotimes_i \langle R_i, m_i\rangle$ *est non-réentrant.*

La définition suivante est simplement une variante de la définition 3.59 tenant compte des symétries.

Définition 3.63 (Pré-ordre bien adéquat) *Soit $\langle S, h\rangle$ le dépliage d'un réseau symétrique $\langle R, \mathcal{S}, m_0\rangle$. Un pré-ordre $\preceq$ sur les configurations du dépliage est bien adéquat si :*

- *$\preceq$ rafine $\sqsubseteq$,*
- *$\preceq$ est bien fondé,*
- *$\preceq$ est compatible avec l'extension : pour toute transition t et toute paire de configurations C_1, C_2 de S telle que $\exists g \in \mathcal{S}, g(h(Cut(C_1))) = h(Cut(C_2))$:*
  1. *$C_1 \prec C_2 \Rightarrow \forall C_2' \in C_2 \cdot t, \exists C_1' \in C_1 \cdot g(t) : C_1' \prec C_2'$*
  2. *$C_1 \equiv_\preceq C_2 \Rightarrow \forall C_2' \in C_2 \cdot t, \forall C_1' \in C_1 \cdot g(t) : C_1' \equiv_\preceq C_2'$*

La proposition suivante généralise la proposition 3.61 pour les produits de réseaux symétriques.

Proposition 3.64 *Soit pour tout i, $\preceq_i$ un pré-ordre bien adéquat sur les configurations des dépliages de réseaux symétriques $\langle R_i, \mathcal{S}_i, m_i\rangle$. Soit les relations $\preceq_{[i]}$ sur les configurations du dépliage du produit $\langle R, \mathcal{S}, m\rangle$ des $\langle R_i, \mathcal{S}_i, m_i\rangle$ définies par $C \prec_{[i]} C'$ si $C[i] \prec_i C'[i])$. Alors ces relations sont des pré-ordres bien adéquats. De plus, si $\langle R, \mathcal{S}\rangle$ est non-réentrant et si pour tout i, $\preceq_i$ is un ordre total alors $lex(\prec_{[1]}, \ldots, \prec_{[n]})$ est un ordre total.*

3.7 Construction modulaire du préfixe d'un produit de réseaux symétriques

Dans cette partie, nous montrons comment construire un préfixe complet basé sur une décomposition modulaire d'un réseau. D'autre part, les symétries sont prises en compte pour réduire la taille du préfixe. Nous poursuivons notre étude à la mise en œuvre d'une implémentation efficace du dépliage pour des systèmes composés de machines à états et de files.

3.7.1 Construction modulaire

L'algorithme 3.1 présente une construction générique d'un préfixe. Les opérations essentielles pour appliquer cet algorithme à un produit de réseaux symétriques sont la manipulation d'un tas à priorité stockant des événements triés par rapport à leur configuration locale, la détection de raccourci et le calcul des extensions d'un préfixe. Pour obtenir une construction modulaire, toutes ces opérations doivent être décomposées en opérations sur les dépliages des composants. Etant données C_i et C_i' deux configurations du dépliage d'un composante i et b_i une condition de i, nous considérons les opérations élémentaires suivantes :

1. décider si $(C_i) \prec (C_i')$,
2. calculer $\widehat{Cut(C_i)}$,

Algorithme 3.1 Unfold

```
Prefix func unfold(PetriNet ⟨R, m0⟩) {
    Prefix prefix(R, m0);
    SortedHeap heap;
    heap.put(prefix.InitEvent());
    while (not heap.isEmpty()) {
        Event event := heap.getMax();
        prefix.addIn(event);
        if (not prefix.isCutoff(event))
            foreach successor in extend(prefix, event)
                heap.put(successor);
    }
    return prefix;
}
```

3. décider si $C_i \cup C'_i$ forme une configuration,
4. décider si $b_i \in Cut(C_i)$.

Les propositions 3.56 et 3.57 établissent que toute condition OU événement du préfixe $v \in B \cup E$ peut être codé par $\Psi(v)$ enrichi du n-uplets des configurations locales sur chaque composant $\langle [v][1], \ldots, [v][n] \rangle$. Pour le calcul d'une extension du préfixe du produit, nous introduisons de nouvelles structures de données permettant de traiter des extensions locales à chaque composant. Pour un composant i, une extension locale l_i est un couple (e_i, Pre_i) avec e_i est un événement du dépliage du comosant i ($e_i \in E_i$) et Pre_i est un ensemble de conditions du dépliage du produit référençant des conditions du dépliage du composant i ($Pre_i \subseteq \Psi^{-1}(B_i)$). Nous imposons que ${}^\bullet e_i = \Psi(Pre_i)$ et que Pre_i est un ensemble de conditions concurrentes du préfixe déjà construit. Notons $\mathcal{L}_i$ l'ensemble des extensions locales et $\mathcal{L}_i(a)$ les extensions étiquetées par l'action $a \in \Sigma_i$ ($\mathcal{L}_i(a) = \{(e_i, Pre_i) \in \mathcal{L}_i \mid \lambda_i(h_i(e_i)) = a\}$). Par convention, nous définissons $\mathcal{L}_i(a) = \{(\epsilon, \emptyset)\}$ pour $a \notin \Sigma_i$. En se basant sur ces structures de données, les opérations élémentaires utilisées pour la construction du préfixe peuvent être réalisées de la manière suivante :

- *Tri des configurations :* Nous utilisons l'ordre lexicographique proposé dans la proposition 3.64.
- *Tester si un événement e est un racourci :* Nous calculons la coupure de la configuration locale de e par l'identité $h(Cut([e])) = \bigcup_i h_i(Cut_R([e][i]))$ (voir la proposition 3.57). Ainsi, il suffit de tester l'existence d'un événement e' ayant la même valeur de coupure et plus petit que e (i.e. $[e'] \prec [e]$).
- *Calcul d'une extension :* Une extension (globale) n'est rien de plus qu'un n-uplet d'extensions locales $\langle l_1, \ldots, l_n \rangle$ relative à une action donnée a ($l_i \in \mathcal{L}_i(a)$). Pour que cette extension soit valide, il suffit de vérifier que les conditions $\bigcup_k Pre_k$ soient concurrentes :
  - $C = \bigcup_k [Pre_k]$ est une configuration. Ce test est réalisé sur chaque composant ($\forall i, \bigcup_k [Pre_k][i]$ est une configuration du préfixe du composant i).
  - Toute condition de $\bigcup_k [Pre_k]$ est dans $Cut(C)$. Ce test est aussi réalisé

sur chaque composant $(\forall i, \forall b \in Pre_i, \Psi(b) \in Cut(C[i]))$.
Noter que les ensembles d'extensions locales $\mathcal{L}_i(a)$ doivent être mis à jour après l'ajout d'un événement dans le préfixe du dépliage du produit.

3.7.2 Produit de machines à états et de files

Appliquer l'algorithme de construction modulaire présenté précédemment pour des types de composants particuliers suppose la donnée d'une représentation du préfixe des composants, ainsi que la réalisation des opérations élémentaires sur leurs configurations. Dans le cas des machines à états (finis) et des files, ces problèmes sont très largement simplifiés. En effet, nous avons une représentation explicite de leurs dépliages.

Le dépliage d'une machine à états est simplement un arbre. Les conditions et les événements sont identifiés par des séquenses de transitions de la machine. La table suivante donne la description formelle du dépliage. Notons S l'ensemble des états de la machine, T l'ensemble de transitions et s_0 l'état initial.

Elèment	Codage
B	$\sigma : \sigma \in T^* \wedge s_0 \xrightarrow{\sigma}$
Min	$\{\epsilon\}$
E	$\sigma : \sigma \in T^+ \wedge s_0 \xrightarrow{\sigma}$

Evénement	${}^\bullet e$	$e^\bullet$
$\sigma \cdot t$	σ	$\sigma \cdot t$

Les configurations du dépliage sont aussi des séquences de transitions. Pour comparer les configurations, nous utilisons la taille de la séquence puis l'ordre lexicographique en considérant une séquence comme un mot sur l'ensemble des transitions. On supposera au préalable que les transitions de la machine sont totalement ordonnées. Notons $\prec_{lex}$ l'ordre lexicographique, $\sqsubset$ la relation "est un préfixe de". Les opérations élémentaires sur ce dépliage sont réalisées de la manière suivante :

1. décider si $(\sigma) \prec (\sigma')$: $|\sigma| < |\sigma'|$ sinon $\sigma \prec_{lex} \sigma'$,
2. calculer $\widehat{Cut(\sigma)}$: $\{s \in S \mid s_0 \xrightarrow{\sigma} s\}$,
3. décider si $\sigma \cup \sigma'$ est une configuration : $\sigma \sqsubset \sigma'$ ou $\sigma' \sqsubset \sigma$,
4. décider si $\sigma'' \in Cut(\sigma)$: $\sigma = \sigma''$.

Considérons une file (non bornée) manipulant un ensemble *Mess*. Les actions représentant l'ajout et le retrait d'un message m seront notées $Send[m]$ et $Receive[m]$. Le dépliage a deux types d'événements : *Send* et *Receive*. Un événement du type *Send* est complètement défini par la liste de messages ajoutés à la file jusqu'à l'action *Send* spécifiée ; de la même manière, un événement du type *Receive* est défini par la liste de messages retirés de la file. Nous considérons trois types de conditions : In, M et Out correspondant aux places du modèle de la file. Nous pouvons identifier ces conditions par leurs uniques événements prédécesseurs. Les conditions In et M sont des sorties des événements du type *Send* et les conditions Out sont les sorties des événements *Receive*. La table suivante donne la description formelle de ce dépliage :

Elément	Codage
B	$In[\sigma_s] : \sigma_s \in Send^*$ $M[\sigma_s] : \sigma_s \in Send^+$ $Out[\sigma_s, \sigma_r] : \sigma_s \in Send^* \wedge \sigma_r \in Receive^* \wedge \mu(\sigma_s) = \mu(\sigma_r)$
Min	$\{In[\epsilon], Out[\epsilon, \epsilon]\}$
E	$Send[\sigma_s] : \sigma_s \in Send^+$ $Receive[\sigma_s, \sigma_r] : \sigma_s \in Send^+ \wedge \sigma_r \in Receive^+ \wedge \mu(\sigma_s) = \mu(\sigma_r)$

Evénement	${}^\bullet e$	$e^\bullet$
$Send[\sigma_s \cdot t_s]$	$In[\sigma_s]$	$In[\sigma_s \cdot t_s]$, $M[\sigma_s \cdot t_s]$
$Receive[\sigma_s \cdot t_s, \sigma_r \cdot t_r]$	$M[\sigma_s \cdot t_s]$, $Out[\sigma_s, \sigma_r]$	$Out[\sigma_s \cdot t_s, \sigma_r \cdot t_r]$

Les configurations sont codées par des couples de séquences de messages $(\sigma_s, \sigma_r) \in Mess^* \times Mess^*$. La première composante représente la liste des messages ajoutés et la deuxième, la liste des messages retirés. Nous imposons que $\sigma_r \sqsubset \sigma_s$. Notons que l'état de la file après la réalisation des événements d'une configuration (σ_s, σ_r) est $\sigma_s \setminus \sigma_r$. De manière plus générale, les opérations élémentaires sur le dépliage sont réalisées de la manière suivante :

1. décider si $(\sigma_s, \sigma_r) \prec (\sigma'_s, \sigma'_r) : |\sigma_s| + |\sigma_r| < |\sigma'_s| + |\sigma'_r|$ sinon $\sigma_s \prec_{lex} \sigma'_s$ ou sinon $\sigma_r \prec_{lex} \sigma'_r$
2. calculer $\widehat{Cut(\sigma_s, \sigma_r)} : \{m \in Mess^* \mid \mu(\sigma_s) = \mu(\sigma_r) \cdot m\}$
3. décider si $(\sigma_s, \sigma_r) \cup (\sigma'_s, \sigma'_r)$ est une configuration :
   - $\sigma_s \sqsubseteq \sigma'_s \wedge (\sigma_r \sqsubseteq \sigma'_r \vee (\sigma'_r \sqsubseteq \sigma_r \wedge \mu(\sigma_r) \sqsubseteq \mu(\sigma'_s)))$ ou
   - $\sigma'_s \sqsubseteq \sigma_s \wedge (\sigma'_r \sqsubseteq \sigma_r \vee (\sigma_r \sqsubseteq \sigma'_r \wedge \mu(\sigma'_r) \sqsubseteq \mu(\sigma_s)))$
4. décider si $b \in Cut(\sigma_s, \sigma_r)$:
   - si $b = In[\sigma''_s]$ ou $b = M[\sigma''_s] : \sigma''_s = \sigma_s$
   - si $b = Out[\sigma''_s, \sigma''_r] : \sigma''_s \sqsubseteq \sigma_s \wedge \sigma''_r = \sigma_r$

Nous pouvons remarquer que toutes ces opérations peuvent être implémentées par des calculs élémentaires sur les mots. Il en est de même pour le cas d'une file bornée. La technique la plus simple est de considérer une file bornée comme la synchronisation de deux files non bornées : une file stockant les messages de la file bornée ; une autre file ne stockant qu'un type de message, comptant et limitant le nombre de messages de la file bornée. Une technique quasiment équivalente est de définir explicitement le dépliage d'une file bornée. Les opérations élémentaires sont de même nature que celles d'une file non bornée.

Esparza et Römer [31] donne un calcul du préfixe fini d'une file bornée. Il a été réalisé pour illustrer leur technique de dépliages de produits synchronisés de machines à états. Ils utilisent la modélisation d'une file bornée donnée par la figure 3.12. Cet exemple pose une question : quel est l'intérêt de notre technique alors qu'il est possible de représenter un produit synchronisé de machine à états et de files bornées par un simple produit de machines à états ? Deux réponses jouent en faveur de notre technique : (1) bien que les modèles doivent avoir un nombre fini d'états, le nombre maximal de messages contenus dans une file n'est pas nécessairement connu à priori ; (2) en général, le préfixe calculé par notre technique est bien plus petit. Notre technique avait été implémenté dans un outil

prototype. Cet outil avait été utilisé pour comparer les deux implémentations d'une file sur un modèle simple de producteur-consommateur communiquant par une file : le producteur envoie une séquence finie de messages. Nous avons pu noter que pour une séquence donnée, la taille du préfixe croît linéairement avec la taille de la file avec la technique de Esparza et Römer, alors qu'elle reste constante avec notre technique. Cet exemple est là pour nous rappeler : (1) que notre premier modèle d'une file bornée n'est pas la modélisation la plus efficace pour analyser des systèmes à files ; (2) que l'introduction de symétries dans le modèle des réseaux de Petri conduit à une modélisation des files bien adaptée aux techniques de dépliages.

3.8 Conclusion

Nos principales contributions dans le domaine du dépliage sont : (1) la conception des premiers algorithmes de vérification de formules LTL$\setminus X$; (2) l'étude de systèmes manipulant des files (non bornées). Le bogue que nous avons détecté dans la méthode de F. Wallner [87] a été corrigé de manière remarquable par J. Esparza et K. Heljanko [29, 30]. Ces travaux ont été notre source d'inspiration pour la notion d'ordre doublement adéquat utilisée pour la détection de cycles. Malgré tous ces travaux, notre objectif est et reste la réalisation d'un outil de vérification de formules LTL comparable à l'outil SPIN.

Jusqu'à présent, beaucoup de prototypes ont été développés pour illustrer les techniques de dépliage et leurs évaluations sur des exemples académiques ont démontré la pertinence de l'approche. Nous pouvons noter toutefois l'outil PEP [5] qui intègre un module de vérification pour les propriétés de sûreté. Cependant, je ne connais pas de grand succès des techniques de dépliages sur des systèmes industriels comme il en existe pour les BDD ou les techniques de réduction par ordre partiel. Je pense que la réalisation d'un outil est à ce jour nécessaire pour faire le point sur ces techniques et je suis convaincu que des améliorations et des optimisations sont à trouver pour rendre la technique apte à traiter des systèmes de grande taille. Par exemple, sur un problème aussi simple que la détection d'un état bloquant, il est nécessaire de construire un préfixe complet avant de lancer l'algorithme de détection alors que dans la plupart des autres méthodes, cette détection est réalisée simplement à la volée.

Chapitre 4

Vérification symbolique

Dans les années 90, les industries des composants électroniques, dans leur recherche d'outils pour améliorer le niveau de confiance de leurs produits, ont adopté les diagrammes de décisions binaires (BDD) [8] pour traiter des composants de plus en plus complexes. Les BDD sont des structures codant des fonctions booléennes. Ils peuvent être vus comme des arbres où les états représentent des choix de valeurs de variables booléennes ; un ordre total sur les variables garantit l'unicité du codage d'une fonction. Les techniques de partage de structures, combinées à des méthodes de réductions, conduisent à des implémentations extrêmement efficaces en pratique [63, 50]. Ainsi, des vérifications exhaustives ont pu être réalisées sur des systèmes comprenant des billions d'états [63, 50]. Le pouvoir d'expression des BDD est suffisant pour manipuler une grande classe de systèmes finis [9]. De plus, certains systèmes dynamiques peuvent être pris en compte avec ce type de techniques [53]. Comme le nombre de variables des systèmes étudiés est un facteur critique, de nombreuses structures "à la BDD" ont été proposées [64, 11]. D'autres structures ont plutôt cherché à étendre le domaine d'application de ces techniques [2, 44, 43, 56, 70, 57].

Dans le cadre de projets industriels, nous avons conçu une nouvelle structure à la BDD, les *Diagrammes de Décisions de Données* (DDD) [17, 7]. Notre objectif était de fournir un outil flexible qui peut être autant que possible adapté pour la vérification de tout type de modèles et qui offre des capacités de traitement similaires aux BDD. A la différence des BDD, les opérations sur nos structures ne sont pas prédéfinies, mais une classe d'opérateurs, appelée *homomorphismes*, est introduite pour permettre à un utilisateur de concevoir ses propres opérations. Dans notre modèle, les variables ne sont pas booléennes ; elles prennent leurs valeurs dans des domaines non nécessairement bornés. Une autre caractéristique intéressante est qu'aucun ordre sur les variables est présupposé dans la définition. De plus, une variable peut apparaître plusieurs fois dans un même chemin. Cette propriété est très utile quand on manipule des structures dynamiques comme les files. Grâce à la grande flexibilité de la structure, nous avons montré l'aptitude des DDD à traiter des programmes VHDL (dans le cadre du projet CLOVIS, un projet DGA). Les succès de notre première étude ont conduit les DDD à être choisis (dans le projet MORSE, un projet RNTL) comme structures pour la vérification de systèmes décrits en LFP [71, 7], un

langage de prototypage de haut niveau.

La principale limite des structures à la BDD est qu'elles ne permettent de traiter que des ensembles finis d'états. En particulier, les DDD ont été conçus pour être une structure généraliste offrant une grande flexibilité pour le codage d'un ensemble d'états et autorisant la conception de nouveaux opérateurs pour simuler une grande variété d'actions. Cependant, dès qu'une vérification nécessite la manipulation d'un ensemble infini d'états, les DDD sont inadaptés. Pendant ma délégation au LSV, dans le groupe *infini*, j'ai conçu une nouvelle structure, les *automates partagés*, pour manipuler des automates finis. L'émergence d'outils à base d'automates pour la manipulation symbolique de structures infinies [91, 26, 3] font qu'améliorer l'efficacité des bibliothèques d'automates est d'un intérêt essentiel. Une approche prometteuse est la réalisation d'une bibliothèque à la BDD parce que les BDD ont largement fait leurs preuves dans la vérification de systèmes finis de grande taille [9]. Notre nouvelle structure pour la manipulation d'automates est basée sur les deux grands principes des BDD :

- (1) une table de hachage garantit une structure canonique forte de la représentation des automates, et stocke une forêt d'automates tout en partageant les sous-structures communes,
- (2) un cache de calcul garde les résultats des opérations sur les automates, ainsi que ceux réalisés sur leurs sous-structures.

Le premier principe offre un test d'égalité sur les automates en temps constant, alors que le deuxième permet d'accélérer l'évaluation des opérations. Le point clef de cette nouvelle structure est la décomposition d'un automate en ses composantes fortement connexes et la conception d'un algorithme de minimisation d'automates incrémental basé sur cette décomposition transformant un automate en un automate partagé. Nous avons expérimentalement comparé PresTaf, une implémentation élémentaire de la logique de Presburger basée sur les automates partagés, et l'outil LASH [91] utilisant des algorithmes classiques sur les automates. Cette étude expérimentale sur un problème d'exploration de l'espace des états de systèmes infinis a montré le grand intérêt de notre nouvelle structure, et plus particulièrement a mesuré le bénéfice que les automates partagés peuvent apporter à des outils comme LASH.

Ce chapitre est organisé en deux parties. La première partie donne les définitions des DDDs, des opérations sur DDD et donne quelques informations sur l'implémentation de la bibliothèque que nous avons réalisée. Cette partie termine par une étude de cas montrant la grande flexibilité des DDD à représenter des modèles complexes. La deuxième partie est dédiée aux automates partagés. Après une étude théorique, nous décrivons les structures de données et les algorithmes pour l'implémentation d'une bibliothèque d'automates partagés. Enfin, nous terminons par une étude expérimentale.

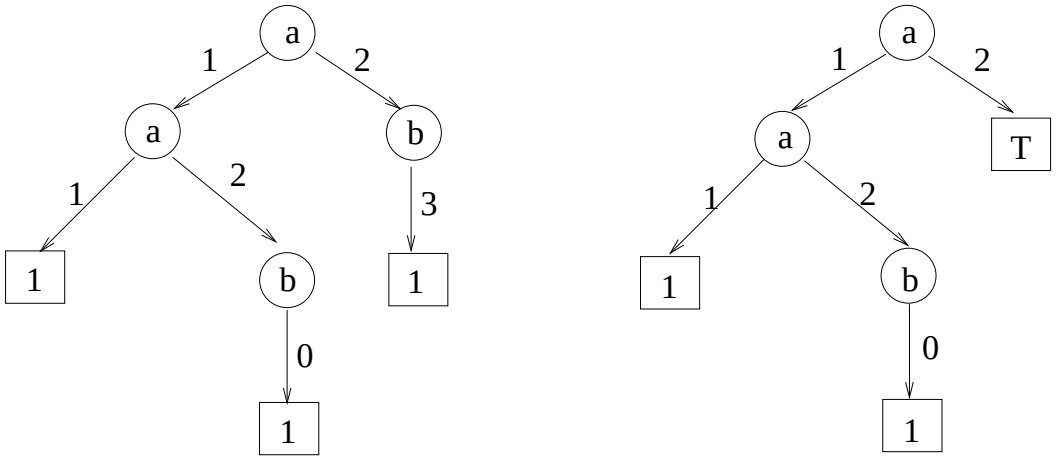


FIG. 4.1 – Deux exemples de Diagrammes de Décisions de données

4.1 Diagrammes de décisions de données

4.1.1 Définitions des DDD

Les diagrammes de décisions de données (DDD) sont des structures de données pour représenter des ensembles de séquences d'affectations de la forme $e_1 = x_1; e_2 = x_2; \cdots e_n = x_n$ où les e_i sont des variables et les x_i des valeurs de variables. Quand les variables apparaissent dans un ordre prédéfini et qu'elles sont de type booléen, les DDD coïncident avec les diagrammes de décisions binaires. Pour les diagrammes de décisions de données, nous n'imposons pas d'ordre, une variable peut apparaître plusieurs fois dans une même séquence et les domaines des variables peuvent être infinis.

Traditionnellement, les diagrammes de décisions sont représentés par des arbres. La figure 4.1 donne à gauche un arbre de décisions contenant l'ensemble des séquences $A = \{(a = 1; a = 1), (a = 1; a = 2; b = 0), (a = 2; b = 3)\}$. Le noeud identifié par 1 est le noeud terminal acceptant, et 0 le noeud non acceptant. Parce que nous ne faisons pas d'hypothèse sur la cardinalité des domaines, les décisions non acceptantes ne sont pas représentées. Ainsi, le noeud 0 n'apparaît pas dans la figure 4.1.

Tout ensemble de séquences ne peut pas être représenté dans notre formalisme. En effet, prenons l'exemple $A \cup \{(a = 2; a = 3)\}$; après l'affectation $a = 2$, on voudrait représenter l'affectation $a = 3$ et $b = 3$ et ceci n'est pas possible dans notre structure : les variables sont les étiquettes des noeuds et deux arcs de même valeur doivent être fusionnées. Nous avons introduit un nouveau type de noeud terminal $\top$, nommé indéfini. Il est utilisé pour donner des approximations de n'importe quel ensemble de séquences. La figure 4.1 donne à droite l'arbre de décisions pour l'ensemble $A \cup \{(a = 2; a = 3)\}$.

Dans la suite, E est un ensemble de variables, pour toute variable e, $\mathrm{Dom}(e)$ représente le domaine de la variable e.

Définition 4.1 (Diagramme de Décisions de Données) *L'ensemble* $\mathbb{D}$ *des DDDs est construit inductivement par* $d \in \mathbb{D}$ *si :*

- $d \in \{0, 1, \top\}$ *ou*
- $d = (e, \alpha)$ *avec :*
  - $e \in E$
  - $\alpha : \mathrm{Dom}(e) \to \mathbb{D}$, *tel que* $\{x \in \mathrm{Dom}(e) \mid \alpha(x) \neq 0\}$ *est fini.*

Nous notons $e \xrightarrow{a} d$, *le DDD* (e, α) *avec* $\alpha(a) = d$ *et pour tout* $x \neq a$, $\alpha(x) = 0$.

Pour avoir une représentation canonique des DDD, nous introduisons une relation d'équivalence.

Proposition 4.2 (Relation d'équivalence) *Soit* $\equiv$ *la relation définie inductivement pour tout* $d, d' \in \mathbb{D}$ *par*

- $d = d'$, *ou*
- $d = 0$, $d' = (e', \alpha')$ *et* $\forall x \in \mathrm{Dom}(e') : \alpha'(x) \equiv 0$, *ou*
- $d = (e, \alpha)$, $d' = 0$ *and* $\forall x \in \mathrm{Dom}(e) : \alpha(x) \equiv 0$, *ou*
- $d = (e, \alpha)$, $d' = (e', \alpha')$ *and* $(d \equiv 0) \wedge (d' \equiv 0)$, *ou*
- $d = (e, \alpha)$, $d' = (e', \alpha')$ *and* $(e = e') \wedge (\forall x \in \mathrm{Dom}(e) : \alpha(x) \equiv \alpha'(x))$

La relation $\equiv$ *est une relation d'équivalence.*

A partir de maintenant, nous identifirons un DDD à sa classe d'équivalence. Nous utiliserons 0 pour représenter l'ensemble vide. Une représentation canonique d'un DDD peut être obtenue en remplaçant tous les DDD équivalents à 0 par le noeud terminal 0.

Une caractéristique importante des DDD est la notion d'approximation. Intuitivement, $\top$ désigne n'importe quel ensemble de séquences. Quand le noeud $\top$ n'apparaît pas dans un DDD, le DDD représente précisément un ensemble de séquences ; nous dirons que le DDD est bien défini.

Définition 4.3 (Bien défini) *Un DDD* d *est bien défini si*

- $d = 0$, *ou*
- $d = 1$, *ou*
- $d = (e, \alpha)$ *où* $\forall x \in \mathrm{Dom}(e) : \alpha(x)$ *est bien défini.*

D'une certaine manière, $\top$ est la pire des approximations d'un ensemble de séquences. Nous introduisons un ordre partiel formalisant la notion d'approximation : la relation "mieux défini que".

Proposition 4.4 (Mieux défini) *Soit la relation* $\preceq$, *appelée* "mieux défini que", *définie pour tout* $\forall d, d' \in \mathbb{D}$ *par* $d \preceq d'$ *si*

- $d' = \top$, *ou*
- $d \equiv d'$, *ou*
- $d \equiv 0$, $d' = (e', \alpha')$ *et* $\forall x \in \mathrm{Dom}(e') : 0 \preceq \alpha'(x)$
- $d = (e, \alpha)$, $d' = (e', \alpha')$ *et* $(e = e') \wedge (\forall x \in \mathrm{Dom}(e) : \alpha(x) \preceq \alpha'(x))$

La relation $\preceq$ *est un ordre partiel sur les classes d'équivalences de* $\mathbb{D}$. *De plus les DDD bien définis sont les éléments minimaux par rapport à la relation* $\preceq$.

Nous imposons que les opérateurs sur les DDD doivent être compatibles par rapport à la relation mieux défini. Ceci conduit à la définition suivante :

Définition 4.5 (Opérateur) *Soit f une fonction de $\mathbb{D}^n \to \mathbb{D}$. f est un opérateur sur $\mathbb{D}$ si f est compatible par rapport à la relation $\preceq$:*

$$\forall (d_i)_i \in \mathbb{D}^n, \forall (d'_i)_i \in \mathbb{D}^n : \left(\forall i : d_i \preceq d'_i\right) \Rightarrow f((d_i)_i) \preceq f((d'_i)_i)$$

4.1.2 Opérations sur les DDD

Dans cette section, nous proposons une généralisation des opérations ensemblistes respectant la compatibilité par rapport à la relation mieux défini. En particulier, si les opérandes sont bien définis, le résultat sera aussi bien défini que possible.

Définition 4.6 (Opérations ensemblistes) *La* somme $+$*, le* produit $*$*, la* différence $\setminus$ *de deux DDD sont inductivement définis par*

$+$	$0 \ \vee (e_2, \alpha_2) \equiv 0$	1	$\top$	$(e_2, \alpha_2) \not\equiv 0$
$0 \ \vee (e_1, \alpha_1) \equiv 0$	0	1	$\top$	(e_2, α_2)
1	1	1	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$
$(e_1, \alpha_1) \not\equiv 0$	(e_1, α_1)	$\top$	$\top$	$(e_1, \alpha_1 + \alpha_2)$ *si* $e_1 = e_2$ $\top$ *si* $e_1 \neq e_2$

$*$	$0 \ \vee (e_2, \alpha_2) \equiv 0$	1	$\top$	$(e_2, \alpha_2) \not\equiv 0$
$0 \ \vee (e_1, \alpha_1) \equiv 0$	0	0	0	0
1	0	1	$\top$	0
$\top$	0	$\top$	$\top$	$\top$
$(e_1, \alpha_1) \not\equiv 0$	0	0	$\top$	$(e_1, \alpha_1 * \alpha_2)$ *si* $e_1 = e_2$ 0 *si* $e_1 \neq e_2$

$\setminus$	$0 \ \vee (e_2, \alpha_2) \equiv 0$	1	$\top$	$(e_2, \alpha_2) \not\equiv 0$
$0 \ \vee (e_1, \alpha_1) \equiv 0$	0	0	0	0
1	1	0	$\top$	1
$\top$	$\top$	$\top$	$\top$	$\top$
$(e_1, \alpha_1) \not\equiv 0$	(e_1, α_1)	(e_1, α_1)	$\top$	$(e_1, \alpha_1 \setminus \alpha_2)$ *si* $e_1 = e_2$ (e_1, α_1) *si* $e_1 \neq e_2$

où pour toute opération $\diamond \in \{+, *, \setminus\}$, $\alpha_1 \diamond \alpha_2$ est l'application de $\mathrm{Dom}(e_1) \to \mathbb{D}$ *définie par* $\forall x \in \mathrm{Dom}(e_1) : (\alpha_1 \diamond \alpha_2)(x) = \alpha_1(x) \diamond \alpha_2(x)$.

L'opération de concaténation correspond à la concaténation de langage. La définition qui suit tient compte des aspects d'approximation.

Définition 4.7 (Opération de concaténation) *Soit d, d' deux DDD. La* concaténation *$d \cdot d'$ est inductivement définie par :*

$$d \cdot d' = \begin{cases} 0 & si\ d = 0 \vee d' \equiv 0 \\ d' & si\ d = 1 \\ \top & si\ d = \top \wedge d' \not\equiv 0 \\ (e, \sum_{x \in Dom(e)} (e \xrightarrow{x} (\alpha(x) \cdot d'))) & si\ d = (e, \alpha) \end{cases}$$

Les opérateurs que nous venons de définir respectent la définition d'opérateurs sur les DDD (définition 4.5). Ils vérifient en partie les propriétés habituelles tel que la commutativité et l'associativité. Cependant, on peut noter que $*$ n'est pas associatif.

Proposition 4.8 (Propriétés des opérations élémentaires) *Les opérateurs $*, +, \setminus, \cdot$ sont des opérateurs sur les DDD. De plus, $*, +$ sont commutatifs et $+, \cdot$ sont associatifs. L'opérateur $*$ n'est pas associatif.*

Grâce aux propriétés de l'opérateur $+$, nous pouvons représenter un DDD (e, α) comme une somme $\sum_{x \in Dom(e)} (e \xrightarrow{x} \alpha(x))$. Notons que cette somme a un nombre fini de termes non nuls. Par extension, tout DDD est une expression définie à partir des constantes 0, 1, $\top$, de concaténations élémentaires $e \xrightarrow{x} d$ et de sommes de DDD.

Exemple 22 *Soient d_A le DDD représenté dans la figure 4.1 à gauche, et d_B celui de droite. Ces DDD sont représentés formellement par :*

$$\begin{aligned} d_A &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} b \xrightarrow{3} 1 \\ d_B &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} \top \end{aligned}$$

Réalisons quelques calculs :

$$\begin{aligned} d_A + a \xrightarrow{2} a \xrightarrow{3} 1 &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} \left(b \xrightarrow{3} 1 + a \xrightarrow{3} 1 \right) \\ &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} \top \\ &= d_B \\ \left(a \xrightarrow{1} 1 * a \xrightarrow{2} 1 \right) * \top &= 0 * \top = 0 \neq a \xrightarrow{1} 1 * \left(a \xrightarrow{2} 1 * \top \right) = a \xrightarrow{1} 1 * \top = \top \\ d_A \setminus d_B &= a \xrightarrow{2} \left(b \xrightarrow{3} 1 \setminus \top \right) = a \xrightarrow{2} \top \\ d_B \cdot c \xrightarrow{4} 1 &= a \xrightarrow{1} \left(a \xrightarrow{1} c \xrightarrow{4} 1 + a \xrightarrow{2} b \xrightarrow{0} c \xrightarrow{4} 1 \right) + a \xrightarrow{2} \top \end{aligned}$$

Notons que le deuxième calcul donne un contre exemple de l'associativité de l'opérateur produit.

En général, les propriétés classiques de distributivité des opérateurs sont rarement vérifiées. Cependant dès que les opérandes et les résultats sont bien définis, ces propriétés restent valides. Dans la proposition suivante, nous précisons ces propriétés en tenant compte de la relation "mieux défini".

Proposition 4.9 (Distributivité faible) *Le produit $*$ et la concaténation $\cdot$ sont faiblement distributives par rapport à la somme $+$; la différence $\setminus$ est faiblement distributive à droite avec la somme $+$:*

$$\forall d_1, d_2, d \in \mathbb{D} : \begin{cases} d * d_1 + d * d_2 \preceq d * (d_1 + d_2) \\ (d_1 \setminus d) + (d_2 \setminus d) \preceq (d_1 + d_2) \setminus d \\ d \cdot d_1 + d \cdot d_2 \preceq d \cdot (d_1 + d_2) \\ (d_1 \cdot d) + (d_2 \cdot d) \preceq (d_1 + d_2) \cdot d \end{cases}$$

4.1.3 Homomorphismes sur les DDD

Notre objectif est de généraliser la notion d'homomorphisme sur les DDD en tenant compte de la notion d'approximation introduite par la constante $\top$. L'identité $f(d_1) + f(d_2) = f(d_1 + d_2)$ doit être réécrite en remplaçant l'égalité par la relation "mieux défini". Notons que grâce à la distributivité faible des opérateurs, nous déduisons que la propriété d'homomorphisme est vérifiée pour les fonctions classiques suivantes : $d * \mathrm{Id}$, $\mathrm{Id} \setminus d$, $\mathrm{Id} \cdot d$, $d \cdot \mathrm{Id}$ où d est un DDD et Id est la fonction identité. Nous imposons en plus qu'un homomorphisme doit être un opérateur sur les DDD et que l'image de 0 est 0.

Définition 4.10 (Homomorphisme) *Une fonction Φ sur les DDD est un* homomorphisme *si $\Phi(0) = 0$ et*

$$\forall d_1, d_2 \in \mathbb{D} : \begin{cases} \Phi(d_1) + \Phi(d_2) \preceq \Phi(d_1 + d_2) \\ d_1 \preceq d_2 \Rightarrow \Phi(d_1) \preceq \Phi(d_2) \end{cases}$$

La somme et la composition de deux homomorphismes sont des homomorphismes.

Proposition 4.11 (Somme et composition) *Soit Φ_1, Φ_2 deux homomorphismes. Alors $\Phi_1 + \Phi_2$, $\Phi_1 \circ \Phi_2$ sont des homomorphismes.*

Jusqu'à présent, nous avons à notre disposition les homomorphismes $d*\mathrm{Id}$ et $\mathrm{Id} \setminus d$ qui donnent la possibilité d'extraire ou de retirer des séquences contenues dans un DDD donné. Les deux homomorphismes $\mathrm{Id} \cdot d$ et $d \cdot \mathrm{Id}$ permettent la concaténation d'un ensemble de séquences à gauche ou à droite. Par exemple, l'ajout à gauche d'une affectation $e_1 = x_1$ est très souvent utilisé et est réalisé par l'homomorphisme de concaténation $e_1 \xrightarrow{x_1} \mathrm{Id}$. Evidemment, nous pouvons combiner des opérateurs à l'aide de la somme et de la composition.

Cependant le pouvoir d'expression de ces homomorphismes élémentaires est très limité ; par exemple, nous ne pouvons définir l'opérateur qui modifie la valeur d'une variable. Une manière simple de créer sa propre fonction Φ est de donner les valeurs de la fonction pour 1 et les DDD de la forme $e \xrightarrow{x} d$; la valeur de la fonction pour le DDD (e, α) pourra être donnée par $\sum_{x \in \mathrm{Dom}(e)} \Phi(e \xrightarrow{x} \alpha(x))$ et on pourra poser $\Phi(\top) = \top$. Une condition suffisante pour que Φ soit un homomorphisme est que les applications $\Phi(e, x)$ définies par $\Phi(e, x)(d) = \Phi(e \xrightarrow{x} d)$ soient des homomorphismes. Par exemple, $inc(e, x) = e \xrightarrow{x+1} Id$ et $inc(1) = 1$ définissent l'homomorphisme incrémentant la valeur de la première variable.

L'introduction de la récursivité dans la description d'un homomorphisme étend très largement le pouvoir d'expression de ces homomorphismes. Par exemple, nous pouvons généraliser notre opération d'incrémentation à l'homomorphisme $inc(e_1)$ qui incrémente la valeur de la variable e_1. Cet homomorphisme est donné par $inc(e_1)(e,x) = e \xrightarrow{x+1} Id$ si $e = e_1$ et sinon $inc(e_1)(e,x) = e \xrightarrow{x} inc(e_1)$. En effet si la première variable du DDD est e_1 alors la valeur de e_1 est incrémentée, sinon l'homomorphisme est appliqué récursivement sur les variables suivantes. La proposition suivante formalise la notion d'homomorphisme récursif.

Proposition 4.12 (Homomorphisme récursif) *Soit I un ensemble d'indices. Soit $(d_i)_{i\in I}$ une famille de DDD. Soit $(\tau_i)_{i\in I}$ et $(\pi_{i,j})_{i,j\in I}$ une famille d'homomorphismes. Supposons que $\forall i \in I$, l'ensemble $\{j \in I \mid \pi_{i,j} \neq 0\}$ est fini. Alors les fonctions récursives $(\Phi_i)_{i\in I}$:*

$$\forall d \in \mathbb{D}, \Phi_i(d) \quad = \quad \begin{cases} 0 & si\ d = 0 \\ d_i & si\ d = 1 \\ \top & si\ d = \top \\ \sum_{x\in \mathrm{Dom}(e), j\in I} \pi_{i,j} \circ \Phi_j(\alpha(x)) + \tau_i(\alpha(x)) & si\ d = (e,\alpha) \end{cases}$$

sont des homomorphismes. De tel homomorphismes sont appelés homomorphismes inductifs. Nous noterons l'expression $\sum_{j\in I} \pi_{i,j} \circ \Phi_j + \tau_i$ par $\Phi_i(e,x)$.

Un homomorphisme inductif Φ est donné par des homomorphismes inductifs $\Phi(e,x)$ et sa valeur $\Phi(1)$. Les deux exemples suivants illustrent le pouvoir de cette notion pour concevoir de nouveaux homomorphismes. Le premier exemple donne la description formelle de l'opérateur d'incrément. Le second exemple est un opérateur de permutation des valeurs de deux variables.

Exemple 23 *L'opérateur d'incrément est donné par :*

$$\begin{array}{lcl} inc(e_1)(e,x) & = & \begin{cases} e \xrightarrow{x+1} Id & si\ e = e_1 \\ e \xrightarrow{x} inc(e_1) & sinon \end{cases} \\ inc(e_1)(1) & = & 1 \end{array}$$

Détaillons l'application de inc sur un DDD :

$$\begin{array}{rcl} inc(b)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) & = & a \xrightarrow{1} inc(b)(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) \\ & = & a \xrightarrow{1} b \xrightarrow{3} Id(c \xrightarrow{3} d \xrightarrow{4} 1) \\ & = & a \xrightarrow{1} b \xrightarrow{3} c \xrightarrow{3} d \xrightarrow{4} 1 \end{array}$$

Exemple 24 *L'homomorphisme $swap(e_1,e_2)$ permute les valeurs des variables e_1 et e_2. Sa description utilise trois autres homomorphismes récursifs : $rename(e_1)$, $down(e_1,x_1)$, $up(e_1,x_1)$. L'homomorphisme $rename(e_1)$ renomme la première variable e_1 ; $down(e_1,x_1)$ affecte la variable e_1 à la valeur x_1 et recopie l'ancienne affectation de e_1 en première position ; $up(e_1,x_1)$ met en seconde position l'affectation $e_1 = x_1$.*

$$
\begin{array}{lcl}
swap(e_1,e_2)(e,x) & = & \left\{ \begin{array}{ll} rename(e_1) \circ down(e_2,x) & si\ e = e_1 \\ rename(e_2) \circ down(e_1,x) & si\ e = e_2 \\ e \xrightarrow{x} swap(e_1,e_2) & sinon \end{array} \right. \\
swap(e_1,e_2)(1) & = & \top \\
 & & \\
rename(e_1)(e,x) & = & e_1 \xrightarrow{x} Id \\
rename(e_1)(1) & = & \top \\
 & & \\
down(e_1,x_1)(e,x) & = & \left\{ \begin{array}{ll} e \xrightarrow{x} e \xrightarrow{x_1} Id & si\ e = e_1 \\ up(e,x) \circ down(e_1,x_1) & sinon \end{array} \right. \\
down(e_1,x_1)(1) & = & \top \\
 & & \\
up(e_1,x_1)(e,x) & = & e \xrightarrow{x} e_1 \xrightarrow{x_1} Id \\
up(e_1,x_1)(1) & = & \top
\end{array}
$$

Le rôle de chaque homomorphisme récursif est illustré par l'évaluation suivante :

$$
\begin{array}{lcl}
swap(b,d)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) & = & a \xrightarrow{1} swap(b,d)(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) \\
 & = & a \xrightarrow{1} rename(b) \circ down(d,2)(c \xrightarrow{3} d \xrightarrow{4} 1) \\
 & = & a \xrightarrow{1} rename(b) \circ up(c,3) \circ down(d,2)(d \xrightarrow{4} 1) \\
 & = & a \xrightarrow{1} rename(b) \circ up(c,3)(d \xrightarrow{4} d \xrightarrow{2} 1) \\
 & = & a \xrightarrow{1} rename(b)(d \xrightarrow{4} c \xrightarrow{3} d \xrightarrow{2} 1) \\
 & = & a \xrightarrow{1} b \xrightarrow{4} c \xrightarrow{3} d \xrightarrow{2} 1
\end{array}
$$

Remarquons que $swap(b,e)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) = a \xrightarrow{1} \top$.

4.1.4 Implémentation d'une bibliothèque de diagrammes de décisions de données

Pour écrire un programme orienté objet manipulant des DDD, un programmeur a besoin d'une hiérarchie de classes traduisant les concepts mathématiques des DDD, des opérations ensemblistes, de la concaténation, d'homomorphismes et d'homomorphismes inductifs. Ces concepts sont traduits dans notre interface par la définition de trois classes (`DDD`, `Hom` and `InductiveHom`) pour lesquelles tous les moyens de construire et de manipuler des DDD sont donnés. En effet, l'objectif de notre travail est la conception d'une bibliothèque simple d'emploi ; ainsi nous avons utilisé la surcharge des opérateurs de C++ pour rendre aussi intuitif que possible l'interface de la bibliothèque.

D'un point de vue théorique, un homomorphisme inductif Φ est un homomorphisme défini par le DDD $\Phi(1)$ et une famille d'homomorphisme $\Phi(e,x)$. Les homomorphismes inductifs ont en commun leurs méthodes d'évaluation et ceci conduit à la définition d'une classe, nommée `InductiveHom`, qui contient la méthode d'évaluation, et donne, en terme de méthodes abstraites, les composantes

d'un homomorphisme inductif ; $\Phi(1)$ et $\Phi(e, x)$. Pour concevoir un homomorphisme inductif, il suffit de dériver la classe `InductiveHom` et d'implémenter les méthodes abstraites $\Phi(1)$ et $\Phi(e, x)$.

L'implémentation de notre interface est basée sur les trois modules suivants :

- Un module de manipulation des DDD : grâce à la technique de la table d'unicité inspirée des BDD, il implémente le partage des sous structures et garantit la canonicité de la structure arborescente des DDD.
- Un module de manipulation des homomorphismes : il manipule les données, ainsi que les méthodes d'évaluation des homomorphismes. La canonicité syntaxique d'un homomorphisme est garantie par une table d'unicité. Nous utilisons la notion de classes dérivées pour représenter de manière uniforme une large classe de type d'homomorphismes.
- Un module de calcul : il fournit la méthode d'évaluation des opérations des DDD, ainsi que celle de l'évaluation des homomorphismes. Pour accélérer les calculs, ce module utilise un cache de calcul pour éviter les évaluations redondantes. L'utilisation d'un cache de calcul ramène la complexité de l'évaluation des opérations ensemblistes à des temps polynomiaux. Comme les homomorphismes inductifs sont conçus par des utilisateurs de la bibliothèque, nous ne pouvons pas connaître à l'avance la complexité de leurs évaluations.

4.1.5 Une étude de cas : le BART de San Francisco

Cette section décrit sommairement l'utilisation des DDD dans le calcul des états d'un modèle de système ferroviaire, en l'occurence le BART de San Francisco, modélisé dans un langage de prototypage de haut niveau **L*f*P**.

Le langage L*f*P.

L*f*P est un langage pivot destiné à fournir une description formelle de systèmes distribués. Les comportements sont exprimés en **L*f*P** au moyen d'automates et supporte une conception orientée objet. En outre, le langage propose les fonctionnalités de haut niveau telles que l'envoi de messages, les appels de procédures distants, l'instanciation dynamique et la destruction d'instances ainsi que les appels de méthodes avec passage de paramètres. La possibilité de traduire une spécification **L*f*P** en une représentation des états sous la forme de DDD, et des transitions sous la forme d'homomorphismes rendent possible la vérification formelle des modèles au moyen de cette représentation compacte. Nous montrons comment ces caractéristiques ont pu être implémentées en utilisant les DDD pour le calcul des états accessibles du système BART.

Un modèle du système ferroviaire BART.

L'architecture UML du système ferroviaire est présentée Figure 4.2. Le modèle est composé de 7 classes :

- Extern_data représente une approximation du monde réel utilisée par la simulation. Cette classe n'est pas representée par un DDD, elle est im-

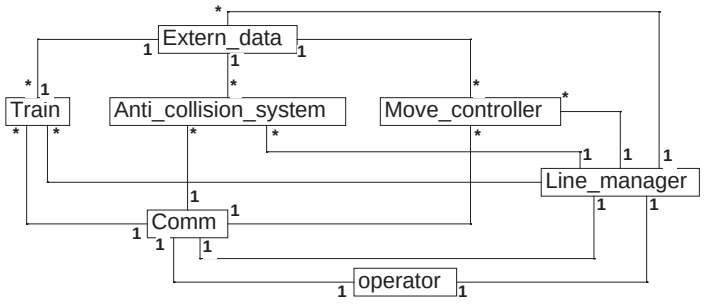


FIG. 4.2 – Description UML du modèle BART

plémentée sous la forme d'une bibliothèque externe modélisant le monde réel.

- Operator représente l'opérateur qui démarre le système et peut remplir d'autres fonctions qui n'ont pas été incluses dans l'implémentation.
- Line_Manager contrôle l'insertion et l'enlèvement des trains sur une ligne.
- Train représente le calculateur à bord d'un train. Celui-ci met à jour les paramètres du train (position, vitesse) en fonction des messages reçus.
- Move_controller et Anti_collision_system sont les contrôleurs associés à un train. Move_controller contrôle les mouvements du train en fonction des paramètres (position, vitesse) ainsi que d'éventuels messages d'alerte en provenance du détecteur de collision Anti_collision_system. Le détecteur de collision est constamment informé de la position du train prédécesseur et peut donc initier les procédures d'urgence.
- Comm représente le système de communication. Il a son propre système d'adressage. Chaque instance communicante reçoit une adresse au début de son exécution. Les messages sont stockés dans des zones mémoires, appelées *binders*. Un binder global partagé par toutes les instances contient les messages envoyés. A chaque instance est associé un binder local et un processus, appelé *média*, qui transfert les messages du binder global vers le binder local.

Chaque instance d'objet s'exécute selon l'automate associé. Le calcul des états se fait en considérant que tout franchissement de transition est atomique. Ce calcul prend donc en compte tous les entrelacements possibles des exécutions parallèles.

Calcul des états accessibles avec les DDD

Nous décrivons dans ce paragraphe le codage du modèle **L***f***P** au moyen des DDD. Schématiquement, la structure choisie pour coder l'état d'un modèle exprimé en **L***f***P** est représentée figure 4.3. *Global information* sont les variables globales. Les variables globales contiennent aussi un ou plusieurs *binders globaux* servant aux communications entre les instances. Un seul binder global est créé dans le cas du BART. Une instance de classe comporte des attributs, un compteur de programme et une pile. Le *binder* est un multi-ensemble servant à modéliser un réceptacle de messages. Le *local média* est une instance par-

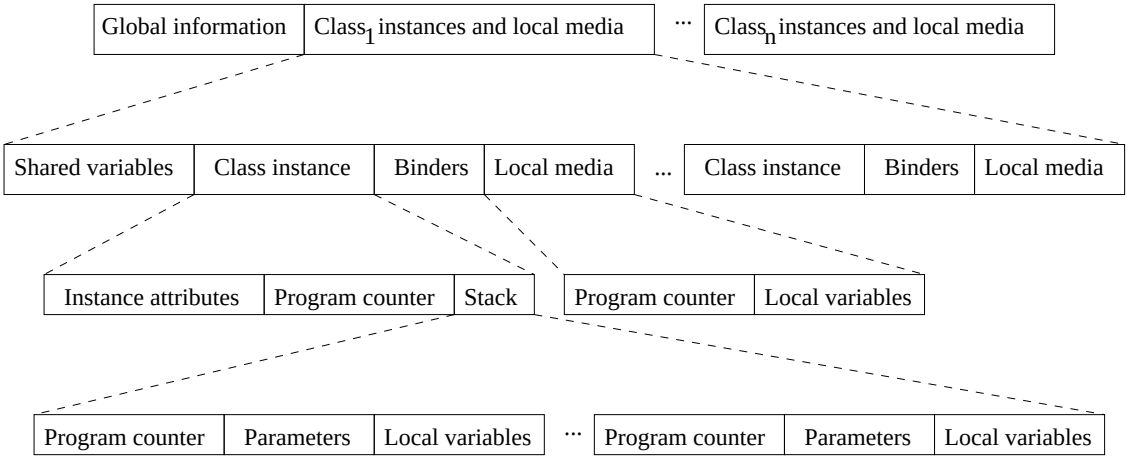


FIG. 4.3 – Structure d'un état du modèle

ticulière modélisant le protocole de communication. La définition d'un ordre déterministe à tous les niveaux de la hiérarchie est très importante pour garantir une représentation canonique de l état. Une étude préliminaire du modèle doit être menée afin de déterminer un codage canonique efficace de l'état.

Dans notre implémentation du système BART, un état a les caractéristiques dynamiques suivantes :

- le nombre d'instances peut varier (création et destruction),
- chaque instance contient une pile pouvant contenir des variables locales, des paramètres d'appel de procédure distant, et une valeur de retour du compteur de programme PC,
- les binders contiennent un nombre variable de messages, leur capacité maximale est fixée par le modèle,
- les média locaux stockent des messages, leur capacité maximale est fixée par le modèle,
- les adresses sont stockées dans des variables de type ensemble,
- les messages sont des vecteurs dont la taille dépend de leur type,

Nous avons implémenté un ensemble d'homomorphismes qui, combinés ensemble, permettent de représenter la sémantique opérationnelle de $\mathbf{L}f\mathbf{P}$. La composition de ces homomorphismes permet de réaliser des actions telles que : l'envoi de message, la réception de message, l'évaluation des préconditions des transitions, les appels de procédures distants et des méthodes locales, la création et la destruction d'instances. Ces homomorphismes s'appuient sur des opérateurs de base tels que : l'affectation de variable, l'évaluation d'expressions, les opérations sur les piles, les multi-ensembles et les tableaux. Les transitions sont implémentées au moyen de deux homomorphismes. *Test_transition(t)* va retourner un DDD identifiant les états qui satisfont la précondition. Comme une transition peut être franchissable simultanément par plusieurs instances, un état sera retourné pour chaque instance afin de pouvoir calculer toutes les séquences possibles. *Fire_transition(t)* va retourner l'ensemble des états obtenus après le franchissement de la transition.

Le calcul des états accessibles est réalisé au moyen d'une boucle qui tente le franchissement de chacune des transitions. L'accumulation des états s'arrête quand aucun nouvel état n'est créé.

Résultats de l'expérimentation

L'implémentation du modèle a requis le codage de plus de 100 transitions et l'élaboration d' états représentant jusqu'à $(6\times t+4)$ tâches si t est le nombre maximum de trains. Les expérimentations ont été réalisées sur un Pentium 4 cadencé à 2.2 Giga hertz et 512M de mémoire.

Trois hypothèses ont été définies afin de pouvoir stocker le résultat en mémoire.

La première hypothèse $H0$ a trait à la mise à jour de la position physique des trains. $H0$ suppose que toutes les transitions procédant à la mise à jour des paramètres physiques des trains sont franchies séparément et ne sont entrelaçables avec aucune autre transition du système. Ces transitions ne sont examinées que lorsque le calcul des états ne progresse plus avec les données courantes. Cette hypothèse est relativement faible dans la mesure où elle permet d'éviter de considérer des cas de détection de collision de deux trains dont les positions sont calculées à des dates différentes.

La seconde hypothèse $H1$ contient $H0$ et suppose que les communications locales entre une instance et son média local sont terminées avant la mise à jour synchronisée de la position des trains. Ceci nous permet d'éviter de considérer des cas d'arrivée tardive de messages asynchrones. Ces messages invalides sont en contradiction avec l'hypothèse que dans le modèle, les communications sont fiables. Cette hypothèse suppose également que le Line_Manager ne tente pas l'ajout ou le retrait d'un train lors de la mise à jour synchronisée des trains. Cette hypothèse vise à limiter l'entrelacement des franchissements des transitions au moment ou la simulation modifie les positions des trains.

La troisième hypothèse $H2$, est beaucoup plus forte et a été mise en place pour permettre le calcul des états avec 512M de mémoire. Cette hypothèse contient $H1$ et, de plus, suppose qu'aucun message ne circule durant la mise à jour simultanée de la position des trains.

Les résultats montrent que l'impact de cette seconde hypothèse est important même quand la taille du binder global est minimale (3 messages). Toutes les exécutions ont été réalisées avec la même description physique des trains et de la voie. Le modèle physique simulant les trains a été ajusté pour que celui-ci génère 6 positions. Dans le cas d'exécution pour deux trains, 2 situations de traitement anti-collision se produisent. Les tableaux suivant résument quelques expérimentations. Le DDD résultant contient tous les états accessibles. Colonne *Taille Glb* est la capacité de stockage, en nombre de messages, du binder global. Colonne *Taille Loc* est la capacité de stockage, en nombre de messages, d'un binder local et de son média. Colonne *NbEtats* est le nombre total d'états. Colonne $LMax$ est la taille maximum de l'état en nombre d'affectations. Colonne *partage* est la taille du DDD en nombre de noeuds. Un noeud utilisé plusieurs fois compte pour 1 en utilisation mémoire. Colonne *sans partage* est la taille du DDD sans utiliser le partage. Une comparaison avec la colonne précédente

donne une idée qualitative du codage de l'état. En effet, un partage efficace est important pour la sauvegarde de l'espace mémoire. Colonne *Temps* donne la durée du calcul en secondes. Les tables montrent l'impact de la capacité des binders et média sur le calcul des états. La table 4.1 considère 1 train avec l'hypothèse $H1$. La table 4.2 considère 2 trains avec l'hypothèse $H2$. La table 4.3 considère 2 trains avec l'hypothèse $H1$. Les résultats montrent que les capacités de stockage locales ont un impact limité. Les paramètres déterminant sont le nombre de trains, la capacité du binder global et les hypothèses restrictives. Les résultats de la table 4.3 n'ont pu être terminés faute de mémoire.

Taille Glb	Taille Loc	NbEtats	LMax	partage	sans partage	Temps (sec)
3	1	10606	118	8343	250405	11
4	1	40099	124	18872	853917	27
5	1	74440	130	29222	1.54e+06	46
3	2	48237	121	10708	994775	17
3	3	62068	121	11320	1.25e+06	21
3	4	63706	121	11165	1.28e+06	21

TAB. 4.1 – Impact de la capacité de communication sur le calcul des états, 1 train et H1.

Taille Glb	Taille Loc	NbEtats	LMax	partage	sans partage	Temps (sec)
3	1	286339	182	101600	1.03e+07	1430
4	1	335827	182	122403	1.24e+07	1874
5	1	347075	182	134551	1.29e+07	2099
3	2	363981	182	97713	1.29e+07	1379
3	3	363981	182	97713	1.29e+07	1378
3	4	363981	182	97713	1.29e+07	1379

TAB. 4.2 – Impact de la capacité de communication sur le calcul des états, 2 trains and H2.

Taille Glb	Taille Loc	NbEtats	LMax	partage	sans partage	Temps (sec)
3	1	2572353	197	207273	7.61e+07	4023
3	2	50765313	197	237542	1.13e+09	8152
3	3	53812153	197	230360	1.21e+09	8313
3	4	53887381	197	227043	1.21e+09	8283

TAB. 4.3 – Impact de la capacité de communication sur le calcul des états, 2 trains and H1.

Nous avons montré qu'il était possible de traduire une spécification de haut niveau d'un système distribué complexe dans le formalisme des DDD afin de calculer les états accessibles du système. Cette expérimentation a montré que

les DDD étaient adaptés à la vérification de modèle comportant des mécanismes complexes :

- instanciation et destruction dynamique d'objets,
- systèmes communicant de façon synchrone et asynchrone et utilisant un adressage dynamique,
- interactions entre tâches exécutées en parallèle,
- appels de procédures distants, appels de méthodes, pile, passages de paramètres,

Cette utilisation des DDD a contribué au déboguages et à la validation du modèle. En particulier, il a été possible de détecter et corriger des problèmes non triviaux de synchronisation et d'adressage du modèle original.

4.2 Automates partagés

4.2.1 Préliminaires

Automates finis déterministes Un *automate fini déterministe* (DFA) sur un alphabet fini Σ est une structure $A = \langle Q, \Sigma, \delta, F\rangle$ où Q est un ensemble fini d'états, $F \subseteq Q$ est l'ensemble des états finaux, et la fonction de transition est donnée par

$$\delta : Q \times \Sigma \to Q.$$

Un DFA enrichi d'un état initial $\langle A, q\rangle$, noté A_q, est appelé *DFA marqué*. La fonction de transition δ est étendue aux mots :

$$\begin{array}{lcl} \delta^* : Q \times \Sigma^* & \to & Q \\ \delta^*(p, \epsilon) & = & p \\ \delta^*(p, xa) & = & \delta(\delta^*(p, x), a) \end{array}$$

où x est un mot et $a \in \Sigma$. Ainsi un DFA marqué A_q accepte un mot x si $\delta^*(q, x) \in F$. Notons $L(A_q)$ l'ensemble des mots acceptés par A_q. Deux états sont *Nerode équivalents* si ils acceptent les mêmes mots : $p \sim q \Leftrightarrow L(A_p) = L(A_q)$. L'équivalence de Nerode est compatible avec la fonction de transition ($p \sim q \Rightarrow \delta(p, a) \sim \delta(q, a)$). La compatibilité induit la définition de l'*automate quotient* : $A_{/\sim} = \langle Q_{/\sim}, \Sigma, \delta_{/\sim}, F_{/\sim}\rangle$ avec $\delta_{/\sim}(p_{/\sim}, a) = \delta(p, a)_{/\sim}$. Notons $L(A_{/\sim_{q_{/\sim}}}) = L(A_q)$ pour tout état q. Un DFA est dit *minimal* si tous les états sont deux à deux non équivalents. Rappelons qu'un DFA marqué A_q est minimal (en terme de nombre d'états) si A est minimal et tout état est accessible à partir de q ; de plus tout automate marqué minimal acceptant le langage $L(A_q)$ est égal à A_q modulo un isomorphisme. Minimiser un automate A consiste à calculer l'automate quotient $A_{/\sim}$. Les automates marqués sont utilisés pour représenter des langages. Minimiser un automate marqué induit une représentation canonique du langage. Les opérations ensemblistes sur les langages sont réalisés en construisant un produit synchronisé d'automates et en calculant l'ensemble des états finaux en fonction de l'opération. Par exemple, étant donnés trois langages L_1, L_2, L_3 associés respectivement aux automates marqués A_{1p}, A_{2q}, A_{3r}, l'automate marqué $A_{(p,q,r)}$ du if-then-else (ite) des trois

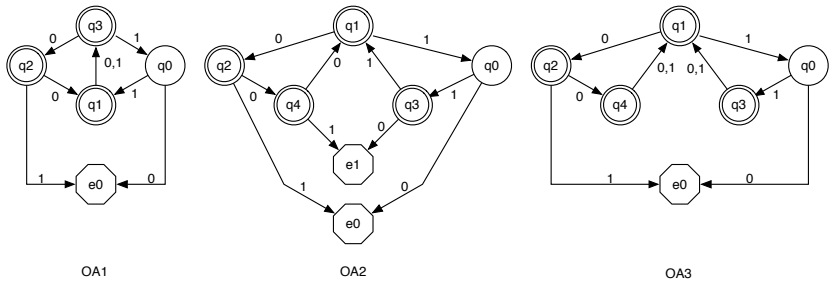


FIG. 4.4 – Exemples d'automates à sorties

langages, $L = \text{ite}(L_1, L_2, L_3) = (L_1 \cap L_2) \cup (\overline{L_1} \cap L_3)$, est défini par :

$$\begin{array}{lcl} Q & = & Q_1 \times Q_2 \times Q_3 \\ \delta((u,v,w),a) & = & (\delta_1(u,a), \delta_2(v,a), \delta_3(w,a)) \\ F & = & \{(u,v,w) : \text{ if } u \in F_1 \text{ then } v \in F_2 \text{ else } w \in F_3\} \end{array} .$$

Deux autres opérations jouent un rôle important pour des calculs symboliques à base d'automates : la clôture existentielle $\text{cl}_a^{\exists}(L)$ et la clôture universelle $\text{cl}_a^{\forall}(L)$ d'un langage L par rapport à une lettre a.

$$\begin{array}{lcl} \text{cl}_a^{\exists}(L) & = & \{w | w.a^* \cap L \neq \emptyset\} \\ \text{cl}_a^{\forall}(L) & = & \{w | w.a^* \subseteq L\} \end{array}$$

L'automate de la clôture existentielle $\text{cl}_a^{\exists}(L(A_q))$ est obtenu en modifiant l'ensemble des états finaux F de l'automates A en $F' = \{p \mid \exists n : \delta^*(p, a^n) \in F\}$. La clôture universelle peut être traitée de manière anologue en tant qu'opération duale de la clôture existentielle : $\text{cl}_a^{\forall}(L) = \overline{\text{cl}_a^{\exists}(\overline{L})}$.

Automates à sorties

Définition 4.13 *Un* automate fini déterministe à sorties *(DFOA) sur un alphabet fini Σ est une structure $\mathcal{O} = \langle Q, Out, \Sigma, \delta, F\rangle$ où Q et Out sont des ensembles finis disjoints d'états, $F \subseteq Q$ l'ensemble des états finaux, et la fonction de transition est donnée par*

$$\delta : Q \times \Sigma \to Q \cup Out$$

Les états de Q sont appelés états locaux*, et les états de Out sont appelés* états de sorties*. Un DFOA enrichi d'un état initial $\langle \mathcal{O}, q\rangle$, noté $\mathcal{O}_q$, est appelé* DFOA marqué*.*

La fonction de transition δ est étendue partiellement à des mots :

$$\begin{array}{lcl} \delta^* : (Q \cup \text{Out}) \times \Sigma^* & \to & Q \cup \text{Out} \\ \delta^*(p, \epsilon) & = & p \\ \delta^*(p, xa) & = & \left\{ \begin{array}{ll} \delta(\delta^*(p,x), a) & \text{si } \delta^*(p,x) \in Q \\ \text{non défini} & \text{sinon} \end{array} \right. \end{array}$$

où x est un mot, et $a \in \Sigma$. Un DFOA marqué $\mathcal{O}_q$ *accepte localement* le mot x si $\delta^*(q,x) \in F$. Notons l'ensemble des mots localement acceptés $L(\mathcal{O}_q)$. Remarquons que quand $\mathcal{O}$ n'a pas de sorties (Out $= \emptyset$), un DFOA est un DFA et que $L(\mathcal{O}_q)$ représente l'ensemble des mots acceptés. Pour tout état de sortie u, nous dirons que x est un *préfixe* de u si $\delta^*(q,x) = u$; l'ensemble des préfixes de u est noté $L[u](\mathcal{O}_q)$. La notion d'équivalence de Nerode s'étend aux états d'un automate à sorties ; deux états sont équivalents si ils acceptent localement les mêmes mots et ont pour toutes sorties les mêmes préfixes : $p \sim q \Leftrightarrow (L(\mathcal{O}_p) = L(\mathcal{O}_q) \wedge \forall u \in \text{Out} : L[u](\mathcal{O}_p) = L[u](\mathcal{O}_q))$. Cette équivalence est compatible avec la fonction de transition. Ainsi l'automate quotient d'un DFOA est donné par $\mathcal{O}_{/\sim} = \langle Q_{/\sim}, \text{Out}, \Sigma, \delta_{/\sim}, F_{/\sim} \rangle$ avec $\delta_{/\sim}(p_{/\sim}, a) = \delta(p,a)_{/\sim}$. Notez que $L(\mathcal{O}_{/\sim_{q_{/\sim}}}) = L(\mathcal{O}_q)$, et $L[u](\mathcal{O}_{/\sim_{q_{/\sim}}}) = L[u](\mathcal{O}_q)$ pour tout état local q. Un DFOA est dit *minimal* si tous les états sont deux à deux non équivalents. Minimiser un DFOA consiste à calculer l'automate quotient $\mathcal{O}_{/\sim}$. La notion de DFOA étend la définition de DFA, mais ne constitue pas immédiatement une structure utile pour notre étude. Seulement les DFOA fortement connexes seront intéressants.

Définition 4.14 *Un DFOA $\mathcal{O} = \langle Q, Out, \Sigma, \delta, F \rangle$ est un DFOA* fortement connexe *(SC-DFOA) si tout état local est accessible à partir de n'importe quel autre état local : $\forall p, q \in Q, \exists x \in \Sigma^* : \delta^*(p,x) = q$.*

Exemple 4.1 La figure 4.4 donne trois exemples d'automates à sorties ; tous sont fortement connexes. Les états locaux sont représentés par des cercles et les états de sorties par des octogones ; un double cercle indique un état final. Alors que *OA1* et *OA2* sont minimaux, la minimisation de *OA3* donne *OA1* (i.e. $q3 \sim q4$). Notez que l'automate quotient d'un SC-DFOA est encore un SC-DFOA.

4.2.2 Automates partagés

Les automates partagés représentent les composantes fortement connexes d'un automate en terme d'automate à sorties et de fonction associant aux sorties des états des autres composantes. La figure 4.5 donne des décompositions d'automates en automates partagés. Dans la suite, nous supposerons que tous les automates sont sur un alphabet donné Σ.

Définition 4.15 *Les* automates partagés *et les* automates partagés marqués *sont inductivement construit de la manière suivante :*

- *si $\mathcal{O}$ est un SC-DFOA sans sortie alors $\mathcal{O}$ est un automate partagé,*
- *si $\mathcal{O}$ est un SC-DFOA sans sortie et q un état local de $\mathcal{O}$ alors $\mathcal{O}_q$ est un automate partagé marqué,*
- *si $\mathcal{O}$ est un SC-DFOA, et λ une fonction associant à toute sortie de $\mathcal{O}$, un automate partagé marqué, alors $\mathcal{O}(\lambda)$ est un automate partagé,*
- *si $\mathcal{O}(\lambda)$ est un automate partagé et q un état local de $\mathcal{O}$, alors $\mathcal{O}_q(\lambda)$ est un automate partagé marqué.*

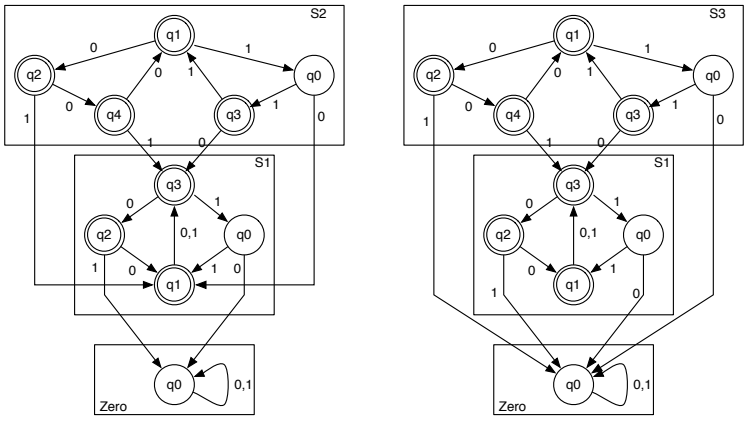


FIG. 4.5 – Décomposition de deux automates en automates partagés

Les fonctions λ sont appelées *connexions*. Notons qu'un automate partagé marqué aurait pu être défini comme un SC-DFOA marqué combiné avec une connexion. Quand un SC-DFOA $\mathcal{O}$ n'a pas de sortie, nous utiliserons la convention $\mathcal{O}(\emptyset)$ pour représenter l'automate partagé. La même convention s'applique aux SC-DFOA marqués sans sortie : $\mathcal{O}_q(\emptyset)$. Notons que cette convention permet de normaliser la notation d'un automate partagé et en particulier tout automate partagé a une connexion. Suivant le contexte, un automate partagé marqué sera considéré comme un automate partagé : par exemple nous parlerons de "l'automate partagé $\lambda(u)$".

Définition 4.16 *L'*ensemble des composantes *d'un automate partagé $\mathcal{S} = \mathcal{O}(\lambda)$, noté $SCC(\mathcal{S})$, est l'ensemble des automates partagés défini inductivement par*

$$SCC(\mathcal{S}) = \{\mathcal{S}\} \cup \bigcup_{u \in Out} SCC(\lambda(u)),$$

où Out est l'ensemble des sorties de $\mathcal{O}$.

Définition 4.17 *La* profondeur *d'un automate partagé $\mathcal{S} = \mathcal{O}(\lambda)$, notée $DEPTH(\mathcal{S})$, est définie inductivement par :*

$$DEPTH(\mathcal{S}) = \begin{cases} 0 & si\ \lambda = \emptyset \\ 1 + Max_{u \in Out} DEPTH(\lambda(u)) & sinon \end{cases}$$

Définition 4.18 *Le* DFA *représentant un automate partagé S, noté $DFA(\mathcal{S})$, est défini par*

- *les états sont les automates partagés marqués $\mathcal{O}_q(\lambda)$ où $\mathcal{O}(\lambda)$ est une composante de $\mathcal{S}$ et q un état local de $\mathcal{O}$,*
- *un état $\mathcal{O}_q(\lambda)$ est final si q est final dans $\mathcal{O}$,*
- *la fonction de transition δ est donnée par :*

$$\delta(\mathcal{O}_q(\lambda), a) = \begin{cases} \mathcal{O}_{\delta(q,a)}(\lambda) & si\ \delta(q,a)\ est\ un\ état\ local \\ \lambda(\delta(q,a)) & sinon \end{cases}$$

Proposition 4.19 *Tout DFA est isomorphe à un DFA représentant un automate partagé.*

Exemple 25 *La figure 4.5 donne plus précisement la représentation graphique de 4 automates partagés :* Zero, S1, S2, S3. *L'automate partagé* Zero *est un automate à sorties sans sortie. Les autres automates sont définis à partir des automates à sorties* $\mathcal{O}1$ *et* $\mathcal{O}2$ *de la figure 4.4 :*

- $\mathcal{S}1 = \mathcal{O}1(\lambda_1)$ *avec* $\lambda_1(e0) = Zero_{q0}$,
- $\mathcal{S}2 = \mathcal{O}2(\lambda_2)$ *avec* $\lambda_2(e0) = \mathcal{S}1_{q1}$, $\lambda_2(e1) = \mathcal{S}1_{q3}$,
- $\mathcal{S}3 = \mathcal{O}2(\lambda_3)$ *avec* $\lambda_3(e0) = Zero_{q0}$, $\lambda_3(e1) = \mathcal{S}1_{q3}$.

Notez que $SCC(Zero) = \{Zero\}$, $DEPTH(Zero) = 0$, $SCC(\mathcal{S}1) = \{Zero, \mathcal{S}1\}$, $DEPTH(\mathcal{S}1) = 1$, *etc.*

Le problème de la canonisation d'un automate partagé nécessite une étude préliminaire sur la caractérisation locale de la minimalité d'un automate. Premièrement, nous donnons une définition de la minimalité d'un automate partagé. Puis, nous en établissons des conditions nécessaires élémentaires. Par la suite, nous en donnerons des conditions suffisantes qui aboutiront à la résolution du problème de canonisation.

Définition 4.20 *Un automate partagé est dit* minimal *si son DFA est minimal.*

Une condition nécessaire évidente de minimalité d'un automate partagé est que tous les automates à sorties composant l'automate soient minimaux.

Proposition 4.21 *Soit* $\mathcal{S}$ *un automate partagé. Soit* $\mathcal{O}(\lambda)$ *une composante de* $\mathcal{S}$. *Si* $\mathcal{S}$ *est minimal, alors* $\mathcal{O}$ *est minimal.*

Une deuxième condition nécessaire est obtenue à partir de la notion d'homomorphisme d'automates partagés. Informellement, un homomorphisme est une fonction reliant les états de deux automates partagés, compatibles avec la fonction de transition.

Définition 4.22 *Soit* $\mathcal{S} = \mathcal{O}(\lambda)$, $\mathcal{S}' = \mathcal{O}'(\lambda')$ *deux automates partagés. Notons* $\mathcal{O} = \langle Q, Out, \Sigma, \delta, F\rangle$ *et* $\mathcal{O}' = \langle Q', Out', \Sigma, \delta', F'\rangle$. *Soit* $h : Q \cup Out \to Q' \cup Out'$ *une fonction vérifiant* $h(Q) \subseteq Q'$ *et* $h(F) \subseteq F'$. *Alors* h *est un homomorphisme de* $\mathcal{S}$ *dans* $\mathcal{S}'$ *si pour toute paire* (p, a) *de* $Q \times \Sigma$, *en posant* $q = \delta(p, a)$ *et* $q' = \delta'(h(p), a)$, *nous avons :*

- *si* $q \in Q$, *alors* $q' \in Q'$ *et* $h(q) = q'$,
- *si* $q \in Out$ *et* $q' \in Out'$, *alors* $\lambda(q) = \lambda'(q')$,
- *si* $q \in Out$ *et* $q' \in Q'$, *alors* $\lambda(q) = \mathcal{S}'_{q'}$.

Nous dirons que $\mathcal{O}$ *est homomorphe à* $\mathcal{O}'$ *si il existe un homomorphisme* $h : \mathcal{O} \to \mathcal{O}'$. *De plus, si* h *est bijective,* $\mathcal{O}$ *et* $\mathcal{O}'$ *sont dits isomorphes.*

La proposition suivante établit qu'un homomorphisme relie des automates marqués reconnaissant les mêmes mots.

Proposition 4.23 *Soient* $\mathcal{S} = \mathcal{O}(\lambda)$, $\mathcal{S}' = \mathcal{O}'(\lambda')$ *deux automates partagés. Soit* $h : \mathcal{S} \to \mathcal{S}'$ *un homomorphisme. Soit* p *un état local de* $\mathcal{O}$. *Alors* $L(\mathcal{S}_p) = L({\mathcal{S}'}_{h(p)})$.

Ainsi, une condition nécessaire de minimalité d'un automate partagé est que ses composantes soient deux à deux non homomorphes.

Proposition 4.24 *Soit $\mathcal{S}$ un automate partagé. Si $\mathcal{S}$ est minimal, alors les automates partagés de $SCC(\mathcal{S})$ sont deux à deux non homomorphes.*

L'automate $S3$ de la figure 4.5 n'est pas minimal bien que les automates à sorties de ses composantes le soient. En effet, $S3$ est homomorphe à $S1$: $h(q0) = q0$, $h(q1) = q3$, $h(q2) = q2$, $h(q3) = h(q4) = q1$, $h(e0) = q3$ et $h(e1) = e0$. Par contre, nous pouvons montrer que $S2$ n'est pas homomorphe à $S1$ et que de plus $S2$ est minimal.

La proposition suivante raffine la propriété d'automates partagés homomorphes. Cette proposition sera très utilisée pour donner une représentation canonique des automates partagés, et aussi pour optimiser nos algorithmes.

Proposition 4.25 *Soient $\mathcal{S} = \mathcal{O}(\lambda)$, $\mathcal{S}' = \mathcal{O}'(\lambda')$ deux automates partagés. Notons Out et Out' les ensembles d'états de sorties de $\mathcal{O}$ et $\mathcal{O}'$, et Q' l'ensemble des états locaux de $\mathcal{O}'$. Si $\mathcal{S}$ est homomorphe à $\mathcal{S}'$, alors*

$$\lambda(Out) \subseteq \lambda(Out') \cup \{\mathcal{S}'_{q'} | q' \in Q'\}$$

De plus si $\lambda(Out) \subseteq \lambda(Out')$ alors $\lambda(Out) = \lambda(Out')$, sinon $DEPTH(\mathcal{S}) = DEPTH(\mathcal{S}') + 1$ et $\forall {\mathcal{S}''}_q \in \lambda(Out) : DEPTH(\mathcal{S}'') = DEPTH(\mathcal{S}') \Rightarrow \mathcal{S}'' = \mathcal{S}'$.

En renforçant les conditions nécessaires de minimalité, nous aboutissons à des règles pour la mise en œuvre d'une représentation canonique des automates partagés. Le point clef est la notion d'ensemble d'automates partagés structurellement minimal.

Définition 4.26 *Soit $\mathcal{F}$ un ensemble d'automates partagés. L'ensemble $\mathcal{F}$ est* structurellement minimal *si les conditions suivantes sont respectées pour toute paire d'automates partagés $\mathcal{S} = \mathcal{O}(\lambda)$ et $\mathcal{S}'$:*

1. *La connexion λ de $\mathcal{S}$ est injective ou vide.*
2. *Posons Out l'ensemble des états de sorties de $\mathcal{O}$. $\mathcal{S}$ n'est homomorphe à aucun automate partagé de $\lambda(Out)$.*
3. *L'automate à sorties de $\mathcal{O}$ est minimal.*
4. *Si $\mathcal{S}$ et $\mathcal{S}'$ sont isomorphes alors $\mathcal{S} = \mathcal{S}'$.*

Les deux propositions suivantes établissent des conditions suffisantes de minimalité et de canonicité des automates partagés.

Proposition 4.27 *Soit $\mathcal{S}$ un automate partagé. Si l'ensemble $SCC(\mathcal{S})$ est structurellement minimal alors $\mathcal{S}$ est minimal.*

Proposition 4.28 *Soient $\mathcal{S}_q$, $\mathcal{S}'_{q'}$ deux automates partagés. Si $SCC(\mathcal{S}) \cup SCC(\mathcal{S}')$ est structurellement minimal, alors les deux propriétés suivantes sont équivalentes :*

1. *$\mathcal{S} = \mathcal{S}'$ et $q = q'$,*
2. *$L(\mathcal{S}_q) = L({\mathcal{S}'}_{q'})$*

```
1  static final int alphabetSize; // Alphabet = {0 ... alphabetSize−1}
2
3  class OutputAutomaton {
4      int        nbLocalStates; // LocalStates  = {0 ...  nbLocalState−1}
5      int        nbOutStates;   // OutStates = {−1 ... −nbOutState}
6      int [ ][ ]    succ;          // transition function
7      boolean [ ] isFinal;      // final state caracteristic function
8  }
9
10 class SharedAutomaton {
11     OutputAutomaton       outputAutomaton;
12     MarkedSharedAutomaton [ ] bindFunction; // image of output state −k
13                                             // is bindFunction[k−1]
14     int   depth;
15 }
16
17 class MarkedSharedAutomaton {
18     SharedAutomaton sharedAutomaton;
19     int  initial ; //  initial  < sharedAutomaton.outputAutomaton.nbLocalStates
20 }
```

FIG. 4.6 – Codage des automates partagés

4.2.3 Implémentation des automates partagés

L'objectif de cette section est de traduire les résultats théoriques de la section précédente, et plus particulièrement la notion d'ensemble structurellement minimal, en une implémentation. Nous avons repris l'idée des tables d'unicité des BDD pour transformer une égalité de structures des automates partagés et à sorties en une égalité de références (ou pointeurs). Transformer l'égalité des automates à sorties modulo une permutation des places locales en une égalité physique est le principal défi de notre implémentation. Nous en proposons une solution simple : une légère modification de l'algorithme de Hopcroft [49, 42].

Représentation canonique forte d'un automate partagé L'alphabet Σ est fixé à $\{0, 1 \cdots |\Sigma| - 1\}$. Les structures élémentaires sont les automates à sorties (SC-DFOA), les automates partagés et les automates partagés marqués. La figure 4.6 donne une définition du codage de ces structures en pseudo-Java.

La méthode `equals` pour les `MarkedSharedAutomaton` a été redéfinie comme une égalité de structure. Deux tables de hachage imposent une canonicité forte des `OutputAutomaton` et `SharedAutomaton` de façon à ce que l'égalité de référence soit équivalente à l'égalité de structure. Ces deux tables sont appelées *table d'unicité des automates à sorties* et *table d'unicité des automates partagés*.

Les tables d'unicité associent à toute structure une référence unique. Ainsi tous les `OutputAutomaton` et les `SharedAutomaton` ont des entrées uniques dans leurs tables. Avant qu'une nouvelle structure soit créée, une recherche dans la table d'unicité détermine si la structure existe déjà. Si, la structure est référencée, sa référence est utilisée. Sinon, une nouvelle structure est créée et rangée dans la table d'unicité. Ce principe est utilisé pour la gestion des deux structures : `OutputAutomaton` et `SharedAutomaton`

La proposition 4.28 donne des contraintes sur les tables d'unicité conduisant

Réf.	nbLocalStates	nbOuputStates	succ	isFinal
$\mathcal{O}_{Zero}$	1	0	[[0,0]]	[0]
$\mathcal{O}_{One}$	1	0	[[0,0]]	[1]
$\mathcal{O}1$	4	1	[[-1,1],[3,3],[1,-1],[2,0]]	[0,1,1,1]
$\mathcal{O}2$	5	2	[[-1,3],[2,0],[4,1],[-2,1],[1,-2]]	[0,1,1,1,1]

TAB. 4.4 – Tables d'unicités des automates à sorties

Réf	bindFunction	depth
$\mathcal{S}_{Zero}$	[]	0
$\mathcal{S}_{One}$	[]	0
$\mathcal{S}1$	$[Zero]$	1
$\mathcal{S}2$	$[(\mathcal{S}1, 1), (\mathcal{S}1, 3)]$	2

TAB. 4.5 – Table d'unicité des automates partagés

à une représentation canonique forte des automates partagés. Nous proposons de les reformuler en tenant compte des problèmes d'implémentation.

1. Tout automate partagé rangé dans la table d'unicité a une connexion injective ou vide. Nous imposons que le tableau d'automates partagés marqués représentant la connexion soit trié lexicographiquement (1) par ordre décroissant par rapport à la profondeur, (2) par rapport à la valeur de la référence sur la structure codant l'automate partagé et (3) par l'entier définissant l'état initial.
2. Tout automate partagé rangé dans la table d'unicité ne doit pas être homomorphe à une image de sa connexion.
3. Tout automate à sorties rangé dans la table d'unicité doit être minimal.
4. Les automates à sorties rangés dans la table d'unicité sont deux à deux non isomorphes modulo une permutation des états locaux.

Notons que nous avons imposé une nouvelle condition à la contrainte 1 : un ordre sur le tableau représentant la connexion d'un automate partagé. Cette contrainte, simple à préserver, a essentiellement deux avantages : (1) d'après la proposition 4.25, tester la condition 2 revient à vérifier si l'automate partagé est homomorphe à la première entrée du tableau codant la connexion ; (2) la condition 4 est réduite à une condition d'isomorphisme sur les automates à sorties modulo une permutation des états.

Exemple 26 *Les tableaux 4.4 et 4.5 donnent le contenu des tables après la canonisation de l'automate partagé $\mathcal{S}2$ de la figure 4.5. Nous avons noté un automate partagé marqué $\mathcal{S}_q$ par la paire $(\mathcal{S}, q)$. Dans notre implémentation, nous avons défini deux constantes : les automates partagés $Zero = \mathcal{S}_{Zero}$ et $One = \mathcal{S}_{One}$. Ils représentent respectivement l'automate ne reconnaissant aucun mot et celui reconnaissant tous les mots binaires. Notez que les tableaux* `bindFunction` *sont triés correctement. Par contre, le choix de l'ordre des états des automates à sorties, qui sera justifié dans la sous-section suivante, est le résultat de l'application de notre adaptation de l'algorithme de Hopcroft.*

Transformer un DFA en un automate partagé Transformer un DFA en un automate partagé est la question centrale de la méthode. L'objectif est de calculer et stocker un automate partagé marqué ayant le même langage que l'automate marqué à transformer et de préserver les contraintes d'intégrité des tables d'unicité. Une description informelle de l'algorithme est donnée dans la figure 4.7. Elle est basée sur la transformation locale des composantes fortement connexes (suivant un ordre topologique), et la préservation de tous les résultats intermédiaires dans un cache de calcul. Le traitement d'une composante est réalisé en trois étapes :

1. La première étape consiste à donner une représentation non canonique d'une SCC en termes d'automate partagé. Notons que les valeurs des connexions sont obtenues à partir des calculs précédents sur les SCC successeurs. La première contrainte d'intégrité sur les automates partagés est garantie en triant correctement le tableau des valeurs des connexions et en évitant que deux entrées du tableau aient la même valeur.
2. L'objectif de la deuxième étape est de tester si l'automate partagé est homomorphe à une des entrées du tableau de connexion, et plus précisément la première. Si un homomorphisme est trouvé, tout état de la SCC est équivalent à un automate partagé marqué (mis sous forme canonique) ; sinon, il faut réaliser la troisième étape.
3. La troisième étape consiste à minimiser l'automate à sorties de l'automate partagé et d'en donner une forme canonique forte.

Dans la description de l'algorithme (voir la figure 4.7), nous ne donnons pas les détails d'implémentation concernant le calcul des SCC et leurs transformations en automates partagés non canoniques (étape 1). Notons que lors du parcours récursif de l'automate partagé, il est inutile de prolonger le calcul quand un état (c-à-d un automate marqué) est déjà dans le cache de calcul, ou bien un état d'un automate partagé déjà référencé dans la table d'unicité. Cette technique jouera un rôle majeur pour l'efficacité de la méthode.

Le test de l'existence d'un homomorphisme (l'étape 2) est très simple : après avoir sélectionné un état local p de l'automate partagé, il suffit de vérifier pour tous les états locaux q de l'automate partagé de la première entrée du tableau de connexion, l'existence d'un homomorphisme associant p à q. Ce test consiste à parcourir l'automate partagé à partir de l'état p, et à calculer de nouvelles valeurs de l'homomorphisme tout en s'assurant de la cohérence du résultat. Notons que la complexité de cet algorithme est $O(\Sigma) \cdot O(n) \cdot O(n')$ où n, n' sont les nombres d'états locaux des deux automates partagés.

L'étape 3 est basée sur une légère modification de l'algorithme de Hopcroft [49, 42, 52]. Notre adaptation (voir la figure 4.8) est généralisée pour traiter des automates à sorties. Cet objectif est réalisé dans les lignes 25-27. Généralement l'ensemble des états finaux induit une partition initiale des classes d'équivalence ; pour les automates à sorties, l'ensemble des prédécesseurs de chaque sortie pour chaque lettre de l'alphabet provoque aussi des raffinements des classes d'équivalences. Malgré tout, l'algorithme proposé est quasiment la copie exacte de l'algorithme original, amélioré par la règle de Gries (voir la ligne 12). Pour résoudre le problème de canonicité forte, nous avons introduit ligne

```
1  HashMap cache = new HashMap();
2
3  MarkedSharedAutomaton canonical(MarkedAutomaton A){
4     for all SCC of A in a topologic order {
5        SharedAutomaton shared;
6        MarkedAutomaton [ ] f;
7
8        Assign to "shared" be a (non canonical) shared automaton representation of SCC
9        Assign to "f" be the isomorphism linking local states of "shared" to states of SCC
10
11       // check if shared is homomorphic to an image of the bind function
12       MarkedSharedAutomaton [ ] g = homomorphicCheck(shared);
13       if (g != null) {
14          for all "p" local state of "shared" {
15             cache[f[p]] = g[p];
16          }
17       }
18       else {
19          // compute a canonical represention of the ouput automaton
20          OutputAutomaton,int [ ] <output,h> = minimize(shared.outputAutomaton);
21
22          SharedAutomaton newShared
23                = unique(new SharedAutomaton(output,shared.bindFunction));
24          for all "p" local state of "shared" {
25             cache[f[p]] = new markedSharedAutomaton(newShared,h[p]);
26          }
27       }
28    }
29    return cache[A];
30 }
```

FIG. 4.7 – Algorithme de transformation d'un DFA marqué en un automate partagé marqué

```
1  int nbClass;     // number of classes of the partition
2  Set [ ] B;       // partition of the local states
3  Set [] L; // L is a set of pairs (i,a) of classes and letter to be traited
4  Set [ ][ ] pred; // for all "s" local state and "a" letter : pred[s][a] ={s':succ[s'][a]=s};
5
6  void refinePartitionWith(Set U) {
7     Set pivot = {i | B[i]∩U ≠ ∅∧B[i]∩U ≠B[i]};
8     sort(pivot);
9     for all "i" in pivot (in the order) {
10       B[i], B[nbClass] = B[i]\U, B[i]∩U;
11       for all "a" letter {
12          if ((i,a) ∉L and 0 < |B[i]| ≤ |B[nbClass]|)
13             L.put(i,a);
14          else
15             L.put(nbClass,a);
16       }
17       nbClass++;
18    }
19 }
20
21 OutputAutomaton,int [ ] adaptedHopcroft(OutputAutomaton output) {
22    nbClass = 1;
23    B [0] = {0, ... , output.nbLocalState−1};
24    refinePartitionwith({final states});
25    for all "u" output state and all "a" letter {
26       refinePartitionwith(pred[u][a]);
27    }
28    while (!L.isEmpty() ) {
29       int, int (i,a) = L.get();
30       refinePartitionWith(⋃_{s∈B[i]} pred[s][a]);
31    }
32    Assign to "minimal" to be the quotient automaton
33    Assign to "h" to be the mapping which associates to each local state its class index
34    return <minimal,h>;
35 }
```

FIG. 4.8 – Algorithme de Hopcroft adapté

8 une instruction de tri de l'ensembles *pivot* pour forcer l'algorithme à fonctionner indépendamment de l'ordre des états locaux. Pour notre objectif, cette modification est bénéfique, car l'application de l'algorithme à deux automates à sorties minimaux isomorphes produit exactement les mêmes automates. Ainsi, appliquer deux fois notre algorithme à un automate à sorties renvoie un automate à sorties minimal fortement canonique ; c'est ce que réalise la fonction *minimize* appelée à la ligne 20 de l'algorithme de canonisation (voir la figure 4.7). Nous ne donnons pas la description des structures de données aboutissant à une complexité de $O(|\Sigma|) \cdot O(n \cdot \lg n)$; cependant, notre modification n'introduit pas de surcoût.

Supposons que les tables de hachage soient parfaites, la complexité de l'algorithme de transformation est $O(|\Sigma|) \cdot O(\mathrm{scc}) \cdot O(n)$ où *scc* est la taille de la plus grande SCC de l'automate à traiter. Notez que si les composantes d'un automate sont plus petites qu'une constante (par exemple, les automates acycliques), l'algorithme a une complexité en temps linéaire. En théorie, une table de hachage n'est jamais parfaite. D'un point de vue pratique, on pourra préférer des structures d'arbres équilibrés pour réduire la complexité dans le pire des cas. Informellement, ce choix conduit à une complexité de l'algorithme de

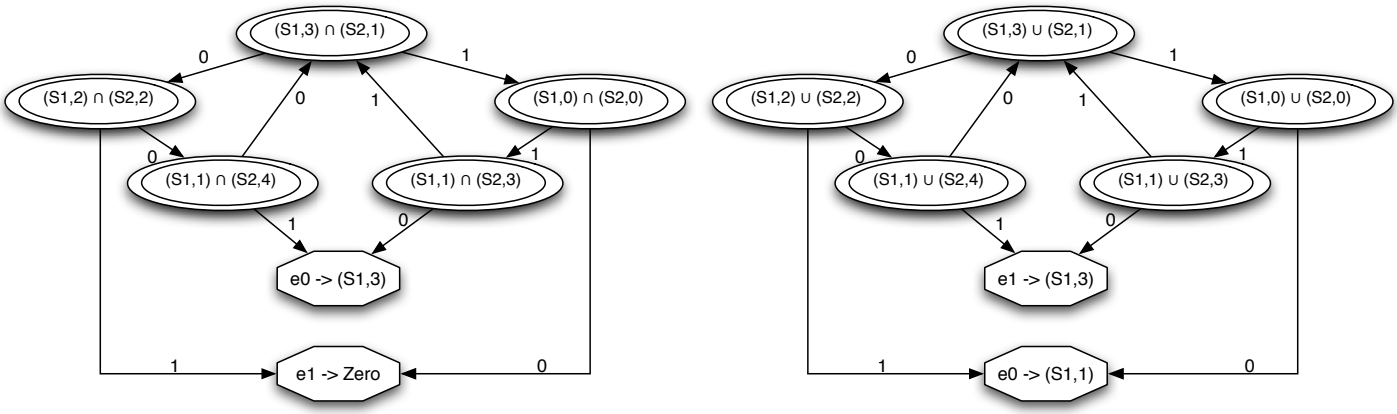


FIG. 4.9 – Exemples de calcul sur les automates partagés

$O(|\Sigma|) \cdot O(n) \cdot O(\lg(n+K))$ où K est une constante tenant compte de la taille des tables d'unicité avant le lancement de l'algorithme.

Exemple 27 *La figure 4.9 donne une représentation graphique des automates partagés (non canoniques) correspondant à l'évaluation des opérations d'intersection et d'union des automates partagés marqués* $(\mathcal{S}1,3)$ *et* $(\mathcal{S}2,1)$*. Notez que nous avons stoppé l'évaluation sur les états identifiés comme des états de sorties. Nous avons simplement utilisé les identités ensemblistes suivantes :* $A \cap \emptyset = \emptyset$, $A \cap A = \emptyset$, $A \cup \emptyset = A$ *et* $A \cup A = A$.

L'algorithme de canonisation pour l'intersection des automates s'arrête par la détection d'un homomorphisme (voir les lignes 12-17 de l'algorithme 4.7). En effet, cet automate est identique à l'automate $\mathcal{S}3$ *de la figure 4.5. Ainsi, avant de renvoyer le résultat de l'évaluation* $(\mathcal{S}1,3)$*, l'algorithme range dans le cache les résultats des sous-calculs suivants :* $(\mathcal{S}1,3) \cap (\mathcal{S}2,1) = (\mathcal{S}1,3)$, $(\mathcal{S}1,2) \cap (\mathcal{S}2,2) = (\mathcal{S}1,2)$, $(\mathcal{S}1,1) \cap (\mathcal{S}2,4) = (\mathcal{S}1,1)$, $(\mathcal{S}1,0) \cap (\mathcal{S}2,0) = (\mathcal{S}1,0)$, $(\mathcal{S}1,1) \cap (\mathcal{S}2,3) = (\mathcal{S}1,1)$.

Par contre, l'algorithme pour l'union des automates va jusqu'à la minimisation. Sur cet exemple, l'algorithme de Hopcroft adapté n'est appliqué qu'une fois parce que l'automate était déjà minimal ; son application n'a fait qu'ordonner canoniquement les états locaux de l'automate à sorties. Notez que ce calcul (voir la ligne 20 de l'algorithme 4.7) renvoie exactement l'automate à sorties $\mathcal{O}1$ *(grâce à l'utilisation de la table d'unicité) et une fonction reliant les états locaux de l'automate à sorties (non canonique) initial à ceux de* $\mathcal{O}1$*. L'automate partagé résultant sera, sur cet exemple, retrouvé dans la table d'unicité des automates partagés (voir les lignes 22-23 de l'algorithme 4.7). Ainsi, avant de renvoyer le résultat de l'évaluation* $(\mathcal{S}2,1)$*, l'algorithme range dans le cache les résultats des sous-calculs suivants :* $(\mathcal{S}1,3) \cup (\mathcal{S}2,1) = (\mathcal{S}2,1)$, $(\mathcal{S}1,2) \cup (\mathcal{S}2,2) = (\mathcal{S}2,2)$, $(\mathcal{S}1,1) \cup (\mathcal{S}2,4) = (\mathcal{S}2,4)$, $(\mathcal{S}1,0) \cup (\mathcal{S}2,0) = (\mathcal{S}2,0)$, $(\mathcal{S}1,1) \cup (\mathcal{S}2,3) = (\mathcal{S}2,3)$.

Implémenter des opérations sur les DFA La méthode de transformation est essentiellement la seule opération utile pour concevoir de nouvelles

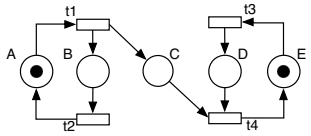


FIG. 4.10 – Modèle de producteur-consommateur

opérations. Nous avons choisi Java pour implémenter notre bibliothèque pour son paradigme objet. `MarkedAutomaton` est une interface dont le contrat est la définition des méthodes `boolean isFinal()` et `MarkedAutomaton succ(int a)`. Dans ce cadre, l'opération ITE est définie par la conception d'une nouvelle classe `IteMarkedAutomaton` implémentant l'interface `MarkedAutomaton`, et réalisée par l'instruction `"A=canonical(new IteMarkedAutomaton(A1,A2,A3))"`. De manière analogue, de nombreuses opérations sur les automates peuvent être conçues. Cependant les opérations de clôture ne rentrent pas dans ce cadre. Nous avons du revoir notre algorithme de transformation en modifiant juste la première étape : l'opération de clôture est réalisée localement à l'automate partagé (non canonique) de l'étape 1. Ainsi deux méthodes ont été ajoutées à notre bibliothèque : `existCloseCanonical` et `forallCloseCanonical`.

4.2.4 Expérimentations

L'objectif est d'évaluer le bénéfice de la technique des automates partagés pour des problèmes de vérification. Nous avons implémenté une bibliothèque élémentaire manipulant des formules de Presburger, appelé PresTaf. Les automates binaires sont utilisés comme structures de données pour coder les formules, ou plus précisément les solutions des formules. Nous ne redonnons pas les techniques de transformations d'une formule de Presburger en un automate. Cette information n'est pas significative pour interpréter les résultats expérimentaux. Nous avons essentiellement suivi les techniques décrites dans [91, 55, 33]. Il est naturel de comparer notre implémentation avec celle de l'outil LASH [91]. En effet, cet outil de référence code les formules par leurs représentations en automates binaires et utilise des algorithmes classiques sur les automates. Ainsi, l'expérimentation mesurera le bénéfice que notre méthode peut apporter à ce type d'outils. Nous avons choisi un problème classique de la vérification pour notre expérimentation : l'exploration en arrière de l'espace d'états d'un réseau de Petri. Ce problème est techniquement intéressant pour les raisons suivantes : la relation de transition d'un réseau de Petri est représentable par une formule de Presburger où les variables sont les places des marquages avant et après franchissement d'une transition ; l'exploration en arrière termine pour certains types d'ensemble de marquages (les ensembles clos par le haut) ; le calcul utilise de manière intensive les opérations manipulant des formules de Presburger. Nous n'avons pas la prétention d'évaluer ou de concevoir la méthode la plus efficace pour traiter ce problème. Les études spécialisées concernant l'exploration de l'espace des états, comme [4], sont en dehors des limites de notre expérimentation.

N	LASH	PresTaf	PresTaf(bis)
2	3s	2s	1s
5	10s	4s	2s
10	29s	10s	3s
20	83s	27s	6s
30	151s	48s	9s
40	-	80s	13s
50	-	122s	19s

sans invariant

N	LASH	PresTaf
50	16s	1s
100	38s	2s
200	89s	3s
500	256s	7s
1000	-	14s
5000	-	67s
10000	-	130s

avec invariant

TAB. 4.6 – Résultats de l'expérimentation pour le modèle du producteur-consommateur

Model	| {Place} |	| {Transition} |	LASH	PresTaf
LEA	30	35	6min 36s	1min 13s
Manufacturing System	14	13	9min 37s	1min 4s
CSM	13	8	14min 38s	1min 2s
PNCSA	31	36	66min	3min22
ConsProd	18	14	1316min	3min55

TAB. 4.7 – Résultats d'expérimentation pour quelques réseaux de Petri

La table 4.6 donne les temps de l'exploration en arrière du modèle de la figure 4.10 avec pour ensemble initial d'exploration $S_N = \{M : M(C) \geq N\}$. Nous avons testé deux types de techniques : une sans optimisation, et une autre où l'exploration est restreinte aux états vérifiant les propriétés invariantes $M(A) + M(B) = 1$ et $M(D) + M(E) = 1$. Les résultats mettent en évidence l'intérêt des automates partagés. D'autre part, appliquer des optimisations est nécessaire pour obtenir des méthodes efficaces. La colonne PresTaf(bis) est un résultat surprenant : une exploration introduit l'évaluation répétée de formules avec un connecteur existentiel sur un ensemble de variables ; le sens commun voudrait qu'évaluer cette formule en une étape (les colonnes LASH et PresTaf) soit plus efficace que de l'évaluer variable par variable. L'intuition ne marche pas sur cet exemple. De tels exemples sont courants quand on utilise des BDD, et ceci souligne la grande similitude entre les automates partagés et les BDD. La table 4.7 confirme les résultats expérimentaux précédents pour d'autres réseaux de Petri. Comme pour les BDD [92], le principal facteur de gain est le cache de calcul. En effet, les itérations de l'exploration d'un espace d'états ont en commun de nombreux sous-calcul. Il semble vraiment que les automates partagés soient un premier pas vers une implémentation efficace des automates à la BDD pour la vérification de systèmes infinis.

4.3 Conclusions

Dans ce chapitre, nous avons décrit une nouvelle structure de données adéquate à la vérification : les diagrammes de décisions de données (DDD). Contrairement aux BDD, ils permettent de traiter les variables définies sur des domaines non bornés, ainsi que des relations de transitions infinies. L'atout de cette nouvelle structure est de fournir la possibilité de définir ses propres opérateurs ou analyses, et ceci afin d'appréhender l'étude de systèmes très variés. De ce travail ont résulté un formalisme bâti sur de solides fondements théoriques et un prototype de bibliothèque DDD. Cette recherche était alimentée par des préoccupations concrètes abordées dans le cadre d'un projet de recherche exploratoire DGA, le projet CLOVIS. Nous avons abouti à un prototype d'outil de vérification de programmes VHDL. Ce projet se poursuit dans le cadre d'un projet de plus grande envergure, un projet RNTL, le projet MORSE : l'objectif est de fournir une méthodologie et des outils prototypes pour le développement d'applications industrielles certifiables dans le domaine des drones. Notre étude s'appuie sur le langage pivot **L***f***P** développé par le LIP6 (Université Paris 6) et les DDD comme structure de données pour la vérification. L'étude de cas que nous avons décrite dans ce chapitre a montré encore une fois l'adéquation des DDD à traiter des systèmes manipulant des structures complexes. Cependant, les performances offertes par notre prototype ne sont pas complètement satisfaisantes. Une nouvelle bibliothèque est en cours de réalisation en se basant sur les nombreuses optimisations développées pour manipuler efficacement les BDD.

Je pense que les perspectives de recherche sur les automates partagés sont beaucoup plus prometteuses. Ces travaux rentrent dans le cadre de la vérification de systèmes infinis. Nous avons conçu la première structure de données pour la manipulation d'automates basée sur les grands principes des BDD : une représentation canonique forte des automates, l'utilisation d'une table d'unicité et d'un cache de calcul. Nos expérimentations montrent le grand bénéfice des automates partagés quand ils sont utilisés pour l'exploration symbolique de l'espace d'états de systèmes infinis. Notre prototype Java (PresTaf) nous a permis de valider nos idées. Une nouvelle bibliothèque, nommé DASH, est en cours de réalisation au LSV pour procéder des expérimentations à grande échelle et être intégrée dans l'outil FAST [3, 55]. Les travaux futurs concernent dans un premier temps à l'application de nos résultats aux techniques la vérification basée sur les automates. Par exemple, les régions d'un automate temporisé sont caractérisées par leurs valeurs d'horloges codées en décimales en base 2 (c-à-d les valeurs codées en binaire par un nombre fini de chiffres après la virgule). Ainsi un ensemble de régions pourrait être représenté par un automate binaire et les opérations sur les régions traduites par des opérations sur les automates. Cette idée serait le point de départ d'une technique de vérification des automates temporisés. Nous pouvons aussi noter les travaux de H. Calbrix [10] sur la représentation des automates de Büchi par des automates de mots finis qui mériterait d'être revus avec le point de vue automates partagés. D'autres perspectives intéressantes sont d'étendre nos résultats à d'autres types de structures comme les automates faibles utilisés par B. Boigelot et P. Wolper [6, 91] pour

implémenter la logique de Presburger sur les réels.

Chapitre 5

Conclusion et perspectives

La vérification est une technologie aux applications pratiques importantes, touchant à des enjeux industriels très larges. C'est aussi un sujet de recherche très actif, qui a donné lieu à une théorie très riche et originale, qui reste toujours très vivante aujourd'hui. Ces recherches se découpent essentiellement en trois grands aspects :

- **Fondamentale :** La vérification s'appuie principalement sur des notions théoriques issues de la logique et de la théorie des automates : deux domaines très riches, en constante progression, et qui ont tendance à se rapprocher ces dernières années.
- **Algorithmique :** Les applications pratiques très importantes donnent tout leur poids à la dimension algorithmique. Ces recherches vont de la théorie de la complexité à la réalisation d'outils de vérification efficaces, capable de traiter des systèmes de plus en plus en complexes.
- **Appliqué :** Ces techniques sont de plus en plus utilisées dans le milieu industriel. Ces recherches vont de la conception de nouveaux langages de description de systèmes à l'intégration des méthodes de vérification dans des méthodologies et des ateliers de Génie Logiciel. L'application des méthodes de vérification n'est plus restreinte au domaine des systèmes informatisés : elle porte aussi sur des domaines aussi divers que la sûreté de fonctionnement de processus industriels et l'analyse de systèmes du monde du vivant.

Dans mon travail, j'ai concentré mes efforts sur l'amélioration de trois aspects de l'algorithmique de la vérification : la logique temporelle linéaire, la vérification par ordre partiel et la vérification symbolique. Pour chacune de ces recherches nous avons établi un bilan et indiqué les poursuites logiques de ce travail. Au-delà, il n'y a guère de doute que l'algorithmique de la vérification va rester longtemps un domaine riche et vivant dans les prochaines années. Parmi les grandes tendances, je noterais trois aspects qui méritent d'être approfondi :

- **Les logiques ordre partiel.** Malgré les nombreux travaux sur les logiques ordre partiel, il n'existe pas d'outil efficace les intégrant. Un travail de mise en œuvre est donc à réaliser. Celui-ci devrait nous conduire sur le choix d'une logique basée sur son pouvoir d'expression mais surtout sur l'efficacité de ses algorithmes de vérification. Rappelons nous qu'il y

a plus de vingt ans, la logique CTL a connu un large succès grâce à la complexité linéaire de sa vérification et son implémentation par des techniques symboliques. La logique LTL est de plus en plus utilisée car on a su maîtriser l'explosion du nombre d'états de l'automate traduisant une formule, mais aussi en partie grâce aux techniques de réduction par ordre partiel. Un travail similaire doit être réaliser pour promouvoir les logiques ordre partiel.

- **La vérification symbolique.** On constate que de nombreux résultats de décidabilité pour les systèmes infinis reposent sur des représentations des données (par exemple, les automates d'arbres), ou des fragments de logique (par exemple, l'arithmétique de Presburger). Par ailleurs, les techniques de représentations symboliques sont encore à ce jour assez limitées. Enrichir ces techniques en se basant sur les idées développées dans des résultats de décidabilité me paraît essentiel.
- **L'analyse de systèmes probabilistes et infinis.** La modélisation de systèmes complexes fait intervenir à la fois des aspects probabilistes et des aspects infinis. Elle concerne non seulement l'étude de systèmes informatisés ou automatiques mais aussi le domaine en plein essor des modèles du vivant. Développer des techniques et des outils de vérification et d'analyse pour ces modèles est d'un intérêt fondamental. Nos travaux sur la vérification de formules LTL sur des systèmes probabilistes étaient motivés par un projet sur l'étude de la propagation de virus dans des populations d'hôtes structurées. Cette étude a mise essentiellement en évidence les besoins des biologistes en matière de modèles et d'outils d'analyse. Les systèmes sont stochastiques et infinis ; les systèmes convergent presque sûrement dans un état sans individu. Jusqu'à présent, seulement des outils de simulations ont pu être utilisés pour l'analyse des modèles. Je pense que dans ce domaine de nouveaux concepts et de nouvelles techniques doivent être élaborés. Par exemple, comment mettre en évidence un phénomène de vague dans la propagation des virus. Il s'agit à la fois d'un phénomène transitoire et dynamique caractéristique de nombreux systèmes du monde vivant.

Bibliographie

[1] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Masson, 1992.

[2] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *ICCAD'93*, volume 2860, pages 188–191, 1993.

[3] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST : Fast Acceleration of Symbolic Transition systems. In *CAV '03*, volume 2725 of *LNCS*, pages 118–121, 2003.

[4] C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *International Journal of Foundations of Computer Science*, 14(4) :605–624, 2003.

[5] E. Best. Partial order verification with PEP. In *POMIV'96*. Am. Math. Soc., 1996.

[6] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Faculté des Sciences Appliquées de l'Université de Liège, 1999.

[7] F. Bréant, J.-M. Couvreur*, F. Gilliers, F. Kordon, I. Mounier, E. Paviot-Adet, D. Poitrenaud, D. Regep, and G. Sutre. Modeling and verifying behavioral aspects. In *Formal Methods for Embedded Distributed Systems : how to master the complexity*, pages 171–211. à paraître chez Kluwer, (* responsable du chapitre), 2004.

[8] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[9] J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic model checking : 10^{20} states and beyond. *Information and Computation (Special issue for best papers from LICS90)*, 98(2) :153–181, 1992.

[10] H. Calbrix. *Mots ultimement péiodiques des langages rationnels de mots infinis*. PhD thesis, Université Paris 7, 1996.

[11] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *ICATPN'2000*, volume 1825 of *LNCS*, pages 103–122. Springer Verlag, 2000.

[12] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1 :275–288, 1992.

[13] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4) :857–907, 1995.

[14] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *FM'99—Formal Methods, Volume I*, volume 1708 of *LNCS*, pages 253–271. Springer Verlag, 1999.

[15] J.-M. Couvreur. Un point de vue symbolique sur la logique temporelle linéaire. In *Actes du Colloque LaCIM 2000*, volume 27 of *Publications du LaCIM*, pages 131–140. Université du Québec à Montréal, 2000.

[16] J.-M. Couvreur. A bdd-like implementation of an automata package. In *soumis à CIAA'04*, 2004.

[17] J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.-A. Wacrenier. Data decision diagrams for petri net analysis. In *ICATPN'02*, volume 2360 of *LNCS*, pages 1–101. Springer Verlag, 2002.

[18] J.-M. Couvreur, S. Grivet, and D. Poitrenaud. Designing a LTL model-checker based on unfolding graphs. In *ICATPN'2000*, volume 1825 of *LNCS*, pages 364–383. Springer Verlag, 2000.

[19] J.-M. Couvreur, S. Grivet, and D. Poitrenaud. Unfolding of product of symmetrical Petri nets. In *ICATPN'2001*, volume 2075 of *LNCS*, pages 121–143. Springer Verlag, 2001.

[20] J.-M. Couvreur and D. Poitrenaud. Model checking based on occurrence net graph. In *FORTE'96*, pages 380–395, 1996.

[21] J.-M. Couvreur and D. Poitrenaud. Detection of illegal behaviours based on unfoldings. In *ICATPN'99*, volume 1639 of *LNCS*, pages 364–383. Springer Verlag, 1999.

[22] J. M. Couvreur, N. Saheb, and G. Sutre. An optimal automata approach to LTL model checking of probabilistic systems. In *LPAR'03*, volume 2850 of *LNAI*, pages 361–375. Springer-Verlag, 2003.

[23] J.M. Couvreur and D. Poitrenaud. Dépliage pour la vérification de propriétés temporelles. In *Vérification et mise en oeuvre des réseaux de Petri - Tome 2*, pages 127–161. Hermes Science, 2003.

[24] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In *CAV'99*, number 1633 in LNCS, pages 172 – 183. Springer Verlag, 1999.

[25] A. Duret-Lutz and R. Rebiha. Spot : une bibliothèque de vérification de propriétés de logique temporelle à temps linéaire. Master's thesis, DEA Systèmes Informatiques Répartis, Université de Paris 6, 2003.

[26] J. Elgaard, N. Klarlund, and A. Moller. Mona 1.x : new techniques for WS1S and WS2S. In *CAV '98*, volume 1427 of *LNCS*, 1998.

[27] J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28 :575–591, 1991.

[28] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23 :151–195, 1994.

[29] J. Esparza and K. Heljanko. A new unfolding approach to LTL model checking. In *ICALP'2000*, number 1853 in LNCS, pages 475–486. Springer-Verlag, 2000.

[30] J. Esparza and K. Heljanko. Implementing LTL Model Checking with Net Unfoldings. In *SPIN'2001*, number 2057 in LNCS, pages 37–56. Springer-Verlag, 2001.

[31] J. Esparza and S. Römer. An unfolding algorithm for synchronous products of transition systems. In *CONCUR'99*, volume 1664 of *LNCS*, pages 2–20. Springer Verlag, 1999.

[32] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *TACAS'96*, volume 1055 of *LNCS*, pages 87–106. Springer Verlag, 1996.

[33] A. Finkel and J. Leroux. Polynomial time image computation with interval-definable counters systems. In *SPIN*, volume 2989 of *LNCS*. Springer, 2004.

[34] Carsten Fritz. Constructing Büchi Automata from Linear Temporal Logic Using Simulation Relations for Alternating Büchi Automata. In *CIAA'03*, volume 2759 of *LNAI*, pages 35 – 48. Springer, 2003.

[35] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV'01*, number 2102 in LNCS, pages 53–65. Springer Verlag, 2001.

[36] J. Geldenhuys and A. Valmari. Tarjan's Algorithm Make On-The Fly LTL Verification More Efficient. In *TACAS '04*, volume 2988 of *LNCS*, pages 205–219. Springer Verlag, 2004.

[37] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV'95*. North-Holland, 1995.

[38] D. Giannakopoulou and F. Lerda. From States to Transition : Improving Translation of LTL Formulae to Büchi Automata. In *FMPA 2000*, volume 2090 of *LNCS*, pages 261–277. Springer Verlag, 2000.

[39] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. Springer Verlag, Berlin, 1996.

[40] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *PSTV'93*, pages 109–124, 1993.

[41] B. Graves. Computing reachability properties hidden in finite net unfolding. In *FSTTCS'97*, volume 1346 of *LNCS*, pages 327–341. Springer Verlag, 1997.

[42] D. Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2 :97–109, 1973.

[43] A. Gupta. *Inductive Boolean Function Manipulation*. PhD thesis, Carnegie Mellon University, 1994.

[44] A. Gupta and A. L. Fisher. Representation and symbolic manipulation of linearly inductive boolean functions. In *ICCAD'93*, pages 111–116, 1993.

[45] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fundamenta Informaticae*, 35 :247–268, 1999.

[46] G. J. Holzmann. An improved protocol reachability analysis technique. *Software, Practice & Experience*, 18(2) :137–161, 1988.

[47] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[48] G. J. Holzmann. *The SPIN MODEL CHECKER*. Addison-Wesley, 2003.

[49] E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[50] H. Hulgaard, P. F. Williams, and H. R. Andersen. Equivalence checking of combinational circuits using boolean expression diagrams. *IEEE Transactions of Computer-Aided Design*, 18(7), 1999.

[51] Knuth and Yao. The complexity of nonuniform random number generation. In *Algorithms and Complexity : New Directions and Recent Results, Ed. J. F. Traub*. Academic Press, 1976.

[52] T. Knuutila. Re-describing an algorithm by Hopcroft. *Theoretical Computer Science*, 250 :333–363, 2001.

[53] T. Kolks, B. Lin, and H. De Man. Sizing and verification of communication buffers for communicating processes. In *ICCAD'93*, volume 1825, pages 660–664, 1993.

[54] R. Langerak and E. Brinksma. A complete finite prefix for process algebra. In *CAV'99*, number 1633 in LNCS, pages 184–195. Springer Verlag, 1999.

[55] J. Leroux. *Algorithmique de la vérification des systèmes à compteurs. Approximation et accélération. Implémentation de l'outil FAST*. Thèse de doctorat, ENS de Cachan, 2003.

[56] L. Mauborgne. Binary decision graphs. In *SAS'99*, volume 1694 of *LNCS*, pages 101–116. Springer-Verlag, 1999.

[57] L. Mauborgne. An incremental unique representation for regular trees. *Nordic Journal of Computing*, 7(4) :290–311, 2000.

[58] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, Boston, MA, 1994.

[59] K. L. McMillan. A technique of state space search based on unfoldings. *Formal Methods in System Design*, 6 :45–65, 1995.

[60] K.L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *CAV'92*, volume 663 of *LNCS*, pages 164–175. Springer Verlag, 1992.

[61] S. Melzer and S. Römer. Deadlock checking using net unfoldings. In *CAV'97*, volume 1254 of *LNCS*, pages 352–363. Springer Verlag, 1997.

[62] Y. Métivier, N. Saheb, and A. Zemmari. A uniform randomized election in trees. In *SIROCCO 10*, volume 17 of *Proceedings in Informatics*, pages 259–274. Carleton Scientific, 2003.

[63] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagrams with attributed edges for efficient boolean function manipulation. In *DAC'90*, pages 52–57. ACM/IEEE, IEEE Computer Society Press, 1990.

[64] A.S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *ICATPN'99*, volume 1639 of *LNCS*, pages 6–25. Springer Verlag, 1999.

[65] M. Mäkelä. Maria : modular reachability analyser for algebraic system nets. In *ICATPN'02*, volume 2360 of *LNCS*, pages 283–302. Springer Verlag, 2002.

[66] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, events structures and domains, part I. *Theoretical Computer Science*, 13(1) :85–108, 1981.

[67] D. Oddoux. *Utilisation des automates alternants pour un model-checking efficace des logiques temporelles linéaires.* PhD thesis, Université Paris 7 (France), 2003.

[68] D. Peled. All from one, one from all : on model checking using representatives. In *CAV'93*, number 697 in LNCS, pages 409–423, Berlin-Heidelberg-New York, 1993. Springer Verlag.

[69] D. Poitrenaud. *Graphes de Processus Arborescents pour la Vérification de Propriétés.* Thèse de doctorat, Université P. et M. Curie, Paris, France, 1996.

[70] F. Reffel. BDD-Nodes Can Be More Expressive. In *ASIAN'99*, volume 1742 of *LNCS*, pages 294–307. Springer Verlag, 1999.

[71] D. M. Regep. *LfP : un langage de spécification pour supporter une démarche de développement par prototypage pour les systèmes répartis.* Thèse de doctorat, Université Paris 6, 2003.

[72] S. Römer. An efficient algorithm for the computation of unfoldings of finite and safe Petri nets (on efficiently implementing McMillan's unfolding algorithm). Technical report, Technishe Universität München, Institut für Informatik, Germany, 1996.

[73] K. Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. In *LPAR'01*, volume 2250 of *LNAI*, pages 39–54. Springer Verlag, 2001.

[74] R. Sebastiani and S. Tonetta. More deterministic vs. smaller Büchi automata for efficient ltl model checking. In *CHARME'03*, volume 2860, pages 126–140. Springer Verlag, 2003.

[75] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logic. *Journal of the Association for Computing Machinery*, 32(3) :733–749, 1985.

[76] F. Somenzi and R. Bloem. Efficient büchi automata from ltl formulae. In *CAV'00*, number 1855 in LNCS, pages 248–263. Springer Verlag, 2000.

[77] R. E. Tarjan. Depth-first search and linear algorithms. *SIAM J. Computing*, 1(2) :146–160, 1972.

[78] H. Tauriainen. *On Translating Linear Temporal Logic into Alternating and Nondeterministic Automata.* Research reports 83, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, 2003.

[79] H. Tauriainen and K. Heljanko. Testing ltl formula translation into büchi automata. *STTT - International Journal on Software Tools for Technology Transfer*, 4(1) :57–70, 2002.

[80] X. Thirioux. Simple and efficient translation from ltl formulas to buchi automata. In *Electronic Notes in Theoretical Computer Science*, volume 66. Elsevier, 2002.

[81] J. D. Ullman, A. V. Aho, and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[82] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, volume 483 of *LNCS*, pages 491–515. Springer Verlag, 1991.

[83] A. Valmari. On-the-fly verification with stubborn sets representatives. In *CAV'93*, number 697 in LNCS, pages 397–408. Springer Verlag, 1993.

[84] M. Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *focs85*, pages 327–338, 1985.

[85] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, 1986.

[86] K. Varpaaniemi and M. Rauhamaa. The stubborn set method in practice. In *Advances in Petri Nets*, volume 616 of *LNCS*, pages 389–393. Springer Verlag, 1992.

[87] F. Wallner. Model Checking LTL Using Net Unfoldings. In *CAV'98*, volume 1427 of *LNCS*, pages 207–218. Springer Verlag, 1998.

[88] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2) :72–99, 1983.

[89] P. Wolper. The tableau method for temporal logic : An overview. *Logique et Analyse*, 110–111 :119–136, 1985.

[90] P. Wolper. Constructing Automata from Temporal Logoc Formulas : A tutorial. In *FMPA 2000*, volume 2090 of *LNCS*, pages 261–277. Springer Verlag, 2000.

[91] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *TACAS'00*, volume 1785 of *LNCS*, pages 1–19, 2000.

[92] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of BDD-based model checking. In *FMCAD'98*, pages 255–289, 1998.