Méthodes et algorithmes pour l'évaluation des performances des systèmes

Anne Benoit

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_/_/_/_/_/_/_/_/_/_/

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : **"Informatique : Systèmes et Communications"**

préparée au Laboratoire Informatique et Distribution
dans le cadre de l'école Doctorale "Mathématiques, Sciences et Technologies de l'Information"

présentée et soutenue publiquement

par

Anne Benoit

le 18 Juin 2003

Méthodes et algorithmes pour l'évaluation des performances des systèmes informatiques à grand espace d'états

---

Directeur de thèse : Brigitte Plateau

JURY

M. ANDRZEJ DUDA	, Président
MME SUSANNA DONATELLI	, Rapporteur
M. JEAN-MICHEL FOURNEAU	, Rapporteur
MME BRIGITTE PLATEAU	, Directeur de thèse
M. WILLIAM J. STEWART	, Examinateur

.

Stochastique. Selon le Grand Dictionnaire d'Oxford, le mot fut créé en 1662, et il est maintenant *rarement utilisé*, ou *périmé*. N'en croyez rien. C'est le Grand Dictionnaire d'Oxford qui est *périmé*, et non la stochastique, car ce terme perd chaque jour de son archaïsme. Son sens primitif est "objectif", ou "but à atteindre", d'où les Grecs ont fait dériver un verbe signifiant "viser une cible" et, par extension métaphorique "réfléchir, penser". Il passa dans la langue anglaise, d'abord comme une manière fantaisiste de condenser "moyens propres à conjecturer", ainsi que le prouve la réflexion de Whitefoot au sujet de sir Thomas Browne en 1712 : "Bien qu'il n'eût point de don de prophétie... il excellait pourtant dans une connaissance qui y touche de fort près, je veux dire la stochastique, grâce à quoi il se trompait rarement au sujet d'événements futurs."

Extrait de *l'homme stochastique*, roman de R. Silverberg.

Remerciements

Je tiens tout d’abord à remercier Brigitte Plateau, sans qui ces travaux n’auraient pu avoir lieu. Jonglant habilement entre son travail de directeur de laboratoire, de professeur et de chercheur, elle a su néanmoins trouver le temps nécessaire pour me guider et me motiver dans ma recherche.

Merci aux membres de mon jury, Andrzej Duda, Susanna Donatelli et Jean-Michel Fourneau pour avoir accepté d’en faire partie. Leurs conseils avisés m’ont permis d’éclairer certains points de la thèse.

J’ai eu le plaisir de travailler avec Billy Stewart et Paulo Fernandes, d’écrire des articles en commun avec eux, et les échanges ont toujours été source d’enrichissement. Grâce à Billy, j’ai de plus découvert quelques subtilités de la langue anglaise, et amélioré la qualité de mon anglais. Merci aussi aux thésards et stagiaires de l’équipe E3 du laboratoire ID, avec lesquels j’ai toujours pu échanger sur divers problèmes.

Dans le cadre du projet Decore, Jean-Marc Vincent et Stéphane Mocanu ont proposé des séminaires très intéressant, et j’ai souvent pu discuter avec eux sur des problèmes un peu techniques, merci pour leur grande disponibilité et leurs nombreux conseils ! J’ai aussi été amenée à être responsable d’un axe de ce projet, et nous avons pu obtenir des financements sans lesquels ces recherches n’auraient pas pu avoir lieu. J’adresse donc un immense merci au projet Decore.

Merci aux chercheurs que j’ai eu la chance de rencontrer durant ma thèse, et avec qui l’échange a été très constructif : Oleg Gusak qui m’a initié aux concepts de "lumpability", Mathieu Le Coz et son interface graphique pour PEPS, et surtout Gianfranco Ciardo et Susanna Donatelli, dont j’ai eu l’honneur de faire la connaissance pour échanger des idées et lancer des projets de collaboration sur nos sujets de recherche très proches.

Il y en a bien d’autres, rencontrés au hasard des cours, des écoles et des conférences. Les rencontres permettent toujours un enrichissement des réflexions personnelles et je remercie donc tous ceux qui ont discuté avec moi et m’ont donné ainsi matière à avancer.

Je tiens également à remercier toute la joyeuse équipe du laboratoire Informatique et Distribution qui ont permis un travail dans une ambiance fort sympathique, avec une attention particulière à tous les "coincheurs" (je ne les cite pas, ils se reconnaîtront sans peine !), sans qui la digestion aurait été un peu plus pénible...

Mes pensées vont enfin et surtout vers tous ceux qui m’ont soutenu moralement, et qui ont su accepter mon caractère pas toujours facile à vivre, parfois ma mauvaise humeur pendant certaines périodes un peu difficiles de ma thèse. Je tiens donc à adresser notamment un grand merci à ma famille, et à tous mes amis musiciens, montagnards, et coincheurs.

J’ai sûrement oublié de citer certains d’entre vous qui m’ont grandement aidé dans mon travail, mais au fond de mon coeur je vous remercie chaleureusement.

Table des matières

CHAPITRE

Introduction

1

La croissance rapide des systèmes et des réseaux entraîne de nos jours des problèmes critiques de performances, avec par exemple des phénomènes d'effondrement, une qualité de service non garantie, etc. Pour garantir les performances de ces systèmes, une analyse fine du comportement est nécessaire afin d'identifier les problèmes et de les résoudre. Cette analyse peut s'effectuer sur des traces d'exécution obtenues à l'aide d'outils spécifiques d'observation (monitoring).

Pour anticiper les problèmes de performance avant la construction d'un premier prototype, il est cependant impératif de prévoir les performances du système avant de l'implémenter dans les conditions optimales, dans un souci de gain en coût et en temps lors du développement du système. Pour ce faire, le système est modélisé, et son analyse indique les problèmes potentiels qui peuvent survenir. De plus, nous pouvons ainsi dimensionner le système de manière à en tirer les meilleures performances possibles pour une architecture donnée.

Il est donc fondamental de disposer de méthodes et d'outils permettant d'analyser et de comprendre le comportement de systèmes complexes, afin de répondre à des questions de performance et de coût.

Ces techniques doivent être applicables tout au long du *cycle de vie* du système, de la spécification à l'exploitation. A chaque stade de vie du système, des techniques de différentes natures peuvent être utilisées. Ainsi, *avant la construction du premier prototype du système*, nous sommes amenés à modéliser le système, ou bien à simuler son comportement. *Une fois le premier prototype construit*, nous pouvons l'observer pour tester les performances du système alors qu'il est soumis à des conditions proches de l'exploitation, mais les résultats sont alors parfois difficiles à interpréter et à extrapoler. Cette approche est utilisée tardivement dans le cycle de vie.

Au niveau des différentes techniques, une première approche informelle, basée sur l'intuition et l'expérience, est nécessaire. Cependant, elle ne se base pas sur une méthodologie et sur des données quantifiées et validées, et des effets secondaires non détectés peuvent apparaître par la suite. Une étude plus systématique et formelle des performances s'avère donc indispensable, par modélisation (début du cycle de vie) puis par observation du comportement (fin du cycle de vie). Si nous utilisons uniquement l'observation, nous serons peut être amené à modifier radicalement un système qui a déjà engendré beaucoup de coûts, car cette technique s'emploie alors que le système a déjà été construit.

Nous nous intéressons pour notre part plutôt à la prédiction des performances sur des modèles, ce qui permet d'intervenir à tous les stades de vie d'un projet. L'utilisation d'un formalisme rigoureux et la prise en compte globale du système permettent une analyse fine dont les résultats peuvent être facilement exploités. Cependant, le système est parfois

difficile à modéliser, d'où la nécessité d'un travail sur les formalismes de modélisation, pour pouvoir appréhender plus facilement et de façon plus précise les systèmes étudiés.

Le principal problème est la taille de l'espace d'états du système, qui dépasse souvent le million d'états. On parle du **problème de l'explosion de l'espace d'états**. Des techniques adéquates doivent alors être utilisées pour pouvoir analyser les performances du système.

> **La problématique** *de cette thèse concerne l'évaluation des performances des systèmes informatiques parallèles et distribués à grand espace d'états. L'objectif est d'améliorer les formalismes existant, ainsi que les méthodes et algorithmes, dans le but de pouvoir traiter des modèles de plus en plus complexes.*

Certains domaines de recherche connexes à l'évaluation de performances sont également confrontés au problème de l'explosion de l'espace d'états, et les solutions développées dans ces domaines peuvent être exploitées pour l'évaluation de performances.

Par exemple, les recherches sur la *vérification* de systèmes visent à analyser certaines propriétés qualitatives des systèmes à grand espace d'états, et les techniques développées sont proches de celles utilisées en évaluation de performances, notamment au niveau des formalismes de modélisation employés.

Ainsi, les diagrammes de décision binaire, qui visent à rendre plus compacte une représentation d'états, sont utilisés à la fois en vérification [9, 63] et en évaluation de performances [58]. Des outils de vérification peuvent être adaptés pour évaluer les performances des systèmes, comme cela a été proposé pour des modèles LOTOS dans [48]. D'un autre côté, nous pouvons également dériver des modèles pour l'évaluation des performances à partir de spécifications pour la vérification, comme par exemple des spécifications UML [69].

Nous reviendrons sur ce lien entre la vérification et l'évaluation de performances dans la conclusion (chapitre 6).

Nous commençons par présenter le processus de modélisation d'un système. Ensuite, nous effectuons un tour d'horizon des différents formalismes de modélisation couramment utilisés en évaluation des performances, puis nous détaillons les méthodes numériques employées pour calculer des indices de performances. Enfin nous précisons les apports de cette thèse par rapport à l'état de l'art, tout en exposant le plan de la thèse.

I. PROCESSUS DE MODÉLISATION

Le modèle est une représentation de la réalité dans un formalisme. Il est développé pour répondre à des questions déterminées et comporte certaines limitations, c'est une abstraction du système réel. Ce *système réel* n'existe pas forcément lors du processus de modélisation, il se peut que ce soit un système que nous cherchons à développer. Pour cela nous désirons effectuer une étude préliminaire de ses performances.

A partir du modèle, nous voulons obtenir des résultats sur le système étudié, c'est l'étape de résolution. Certains résultats peuvent nous amener à modifier notre vision du système, et nous inciter à calculer de nouveaux résultats sur le modèle. Pour cela, nous pouvons être amené à complexifier le modèle pour rajouter des informations. De même, nous pouvons nous rendre compte que certains résultats ou éléments du modèle sont inutiles pour calculer les indices de performances recherchés, et nous pouvons ainsi parfois simplifier le modèle pour faciliter sa résolution.

Pour finir, nous comparons les résultats obtenus avec le comportement du système réel (comportement espéré si le système n'est pas encore créé), et nous pouvons alors concevoir le système qui donne les performances optimales.

Ce processus de modélisation est schématisé dans la figure 1.1.

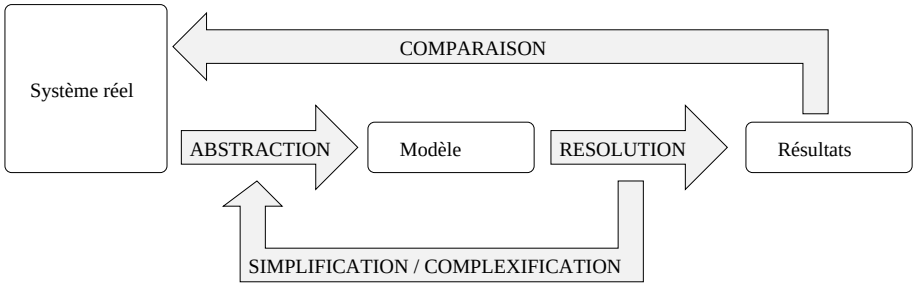


FIG. 1.1 – Processus de Modélisation

Nous détaillons dans la section suivante la phase d'abstraction du système, en présentant différents formalismes de modélisation.

II. FORMALISMES DE MODÉLISATION

Devant la complexité des nouvelles générations de systèmes à concevoir, plusieurs techniques de modélisation ont été développées. Toutes sont basées sur le même principe, consistant a définir successivement :

- les **états** du système (on suppose que les systèmes informatiques étudiés ont un nombre fini d'états) ;
- les **transitions** entre chaque état (dynamique du système) ;
- la **temporisation** des transitions.

Ainsi, les modèles permettent d'analyser le comportement dynamique d'un système en spécifiant l'ensemble de toutes les transitions possibles entre les différents états du système. Les systèmes étudiés s'exécutent dans un environnement aléatoire. Lors de la modélisation d'un tel système, la principale difficulté consiste à modéliser les délais, à savoir le temps que l'on passe dans chaque état en fonction des statistiques sur l'environnement. Dans la plupart des cas, nous supposons que le système est Markovien, donc sans mémoire. L'état futur ne dépend alors que de l'état présent, et non du passé. Cette hypothèse est valide pour la plupart des systèmes étudiés, quitte à compliquer un peu le modèle pour introduire une mémoire (états et transitions supplémentaires).

Les chaînes de Markov, que ce soit à temps continu ou à temps discret, facilitent donc l'analyse des performances des systèmes dynamiques dans de nombreux domaines d'application [89, 49, 44, 40, 74], et elles sont particulièrement bien adaptées à l'étude de systèmes parallèles et distribués [1, 48, 88, 96].

Cependant, lorsque le nombre d'états est important (de l'ordre du million d'états), il est quasiment impossible d'envisager une modélisation à plat, à la fois pour la phase d'abstraction du système, mais aussi en prévision des techniques d'analyse que nous désirons appliquer pour calculer des indices de performances sur le système.

Pour cela, un formalisme de haut niveau structuré est souvent défini, et le modèle sous-jacent est une chaîne de Markov. Un logiciel est alors utilisé pour générer l'espace d'états et le générateur infinitésimal de la chaîne de Markov, ainsi que pour calculer les solutions stationnaires et transitives.

Plusieurs formalismes de haut niveau ont été proposés pour permettre la modélisation de chaînes de Markov très grandes et très complexes de façon compacte et structurée. Parmi ceux largement utilisés dans divers domaines d'application, nous distinguons quatre classes principales de modèles, qui ont chacun des objectifs différents à la base :

- les **réseaux de files d'attente** : approche orientée "ressources consommées par des clients" ;
- les **réseaux de Petri** : analyse fine des synchronisations ;
- les **algèbres de processus** : composition concurrente, exécution parallèle ;
- les **réseaux d'automates** : intégration des synchronisations au modèle état-transition.

Pour adapter le modèle aux systèmes à grand espace d'états, il est possible de structurer davantage le modèle, en introduisant par exemple de la composition ou de la hiérarchie dans ces formalismes. Au niveau de la temporisation, plusieurs approches peuvent également être envisagées, notamment pour différentier les systèmes à temps continu des systèmes à temps discret. Nous trouvons ainsi toute une série de variantes sur ces formalismes de base, et nous n'en citerons que quelques uns.

- Pour le formalisme des *réseaux de files d'attente*, on citera les les réseaux de files d'attente hiérarchiques [13], qui utilisent l'algèbre de Kronecker pour structurer le modèle.
- Au niveau des *réseaux de Petri*, les premières approches ont consisté à introduire des temporisations (réseaux de Petri temporisés, [51]). Les réseaux de Petri stochastiques [61, 25] ont ensuite été définis pour introduire le comportement probabiliste des systèmes. Dans le cas des systèmes à temps discret, on note l'utilisation de réseaux de Petri stochastiques à temps discret [98, 27, 99].
  Au niveau de la structuration des modèles, on a introduit entre autre les réseaux de Petri stochastiques bien définis [33], les réseaux de Petri stochastiques généralisés [73], et les réseaux de Petri stochastiques généralisés superposés [39, 68].
  On trouve également toute une série de variantes sur les réseaux de Petri, par exemple les réseaux stochastiques à récompense [78] ou les réseaux d'activité stochastiques [85].
- Les *algèbres de processus* stochastiques [59, 60] permettent de modéliser des délais dans les algèbres de processus, et sont donc particulièrement bien adaptées à l'évaluation des performances.
- Enfin, les *réseaux d'automates* stochastiques (SANs) [80, 41, 47] permettent à la fois de structurer l'approche par automates et de modéliser les délais. Nous trouvons des formalismes à temps continu et à temps discret. Nous reviendrons en détail sur ce formalisme dans le chapitre suivant, vu que nous basons nos travaux sur ce formalisme.

Les approches sous forme d'algèbre de processus et de réseaux d'automates n'ont pas de notion d'*entité* et de *flot* comme c'est le cas dans les réseaux de files d'attente et dans les réseaux de Petri (clients ou jetons qui circulent). En revanche, elles offrent une vision structurée de sous-systèmes indépendants qui interagissent entre eux (modèles modulaires).

Il est ainsi possible de générer la chaîne de Markov qui représente le système à étudier à partir de formalismes qui modélisent le système entier à partir de sous-systèmes interconnectés.

> *Dans le cadre de cette thèse, nous nous intéressons plus particulièrement au formalisme des* **réseaux d'automates stochastiques** *(SANs).*

Nous ne rentrerons pas dans une discussion détaillée sur une comparaison des différents formalismes. Dans le chapitre 4, nous donnerons quelques éléments de comparaison en introduisant les méthodes de résolution sur les différents formalismes, et donc les différentes façons de stocker les données du modèle. Nous reviendrons sur une comparaison un peu plus approfondie dans la conclusion (chapitre 6, section II.2).

Quel que soit le formalisme utilisé, nous cherchons à calculer des indices de performance sur le système étudié. Certaines méthodes de résolution sont cependant spécifiques à un formalisme donné, mais des techniques peuvent être adaptées d'un formalisme à un autre.

Nous effectuons maintenant un tour d'horizon des différentes méthodes de résolution, sans se placer dans le cadre d'un formalisme en particulier. Nous nous limitons en revanche à l'étude des systèmes markoviens.

III. LES MÉTHODES DE RÉSOLUTION

Les indices de performances qui nous intéressent sont obtenus à partir du vecteur des probabilités stationnaires $\pi \in \mathbb{R}^{|S|}$, où S est l'espace d'états du système Markovien étudié. Le i-ème élément du vecteur π_i représente alors la probabilité que le système soit dans l'état i, au bout d'un temps suffisamment long (toute influence liée à l'état initial doit disparaître).

Nous voulons donc calculer le vecteur π, et c'est la solution du système d'équations linéaires $\pi Q = 0$, avec $\pi e = 1$, où Q est le générateur de la chaîne de Markov, et e est un vecteur dont tous les éléments sont égaux à 1.

Il existe de multiples méthodes numériques de résolution de systèmes linéaires [70, 84, 89]. Cependant, les propriétés liées au générateur infinitésimal d'une chaîne de Markov empêchent l'application de certaines méthodes, et en favorisent d'autres.

Les méthodes numériques dites *directes*, comme la méthode de Gauss [70], ne sont pas utilisables pour des modèles de grande taille. En effet, les besoins en espace mémoire sont trop importants. Or nous nous intéressons justement avec les SANs à l'étude de grands modèles (plusieurs millions d'états). Les méthodes *itératives* sont mieux adaptée à notre cas puisqu'elles permettent de tirer avantage des techniques de stockage creux du générateur infinitésimal. Cependant, là aussi, les besoins en mémoire peuvent devenir trop élevés pour les modèles à l'échelle réelle.

Au niveau des méthodes employées, nous nous intéressons tant à des méthodes simples (méthode de la puissance, Jacobi, Gauss-Seidel) qu'à des méthodes plus complexes comme les méthodes de projection (Arnoldi, GMRES, etc.). Des descriptions de ces méthodes peuvent être trouvées dans [70, 84, 89, 35]. L'opération de base des méthodes itératives est la multiplication du vecteur par le générateur, et ces méthodes ont souvent besoin de calculer ce produit un grand nombre de fois. Leur efficacité dépend donc directement de la façon dont ces données sont stockées.

> *Dans le cadre de cette thèse, nous étudions les systèmes à grand espace d'états. Nous employons donc uniquement des* **méthodes itératives***. Nous reviendrons en détail sur les différentes approches proposées pour la multiplication d'un vecteur de probabilité par le générateur de la chaîne de Markov dans le chapitre 4.*

Nous présentons maintenant le plan de cette thèse, en précisant nos apports par rapport à l'état de l'art.

IV. PLAN DE LA THÈSE

La thèse peut être divisée en deux parties principales. La première regroupe les chapitres 2 et 3, elle se concentre sur le formalisme des réseaux d'automates stochastiques et présente des nouveautés sur ce formalisme. Dans le processus de modélisation (figure 1.1), cette partie concerne la phase d'**abstraction**.

La deuxième partie traite de l'aspect **résolution**, en mettant l'accent sur les algorithmes de multiplication d'un vecteur par le générateur d'une chaîne de Markov (chapitre 4), et présente des résultats numériques (chapitre 5).

Nous détaillons ici le contenu de chacune de ces parties.

IV.1. MODÉLISATION À L'AIDE DE RÉSEAUX D'AUTOMATES STOCHASTIQUES

Le chapitre 2 commence par donner les bases de l'algèbre tensorielle classique et généralisée. Nous présentons ensuite le formalisme SAN, tout d'abord pour le cas des modèles à **temps continu**. Ce formalisme, déjà largement développé depuis une vingtaine d'années [80, 82, 41], décrit le système par le biais de sous-systèmes indépendants qui peuvent interagir. Ces sous-systèmes sont des automates traditionnels auxquels on rajoute des comportements stochastiques et des mécanismes de synchronisation. La matrice de transition de la chaîne de Markov peut alors être exprimée sous un format tensoriel.

Par rapport aux présentations précédentes des SANs à temps continu [41, 42, 43], nous changeons la façon de voir les SANs en mettant l'accent sur la *notion d'événements*. Dorénavant, toutes les transitions des automates sont étiquetées par des événements, et les informations concernant ces événements sont indiquées en dehors du graphe (taux de franchissement, ...). Ceci a pour but d'une part de *clarifier la présentation*, et d'autre part à préparer une *extension* simple et cohérente pour traiter des modèles à temps discret.

IV.1.1. AGRÉGATION TEMPORELLE

En ce qui concerne les modèles à **temps discret**, il s'agit d'un niveau d'abstraction supérieur par rapport aux modèles aux temps continu : le temps est regroupé en intervalles. Ceci permet de réduire la complexité du système car on ne regarde plus le détail de tout ce qui se passe à chaque instant, mais l'ensemble de tout ce qui s'est passé dans un intervalle de temps donné.

Cependant, les modèles à temps discret sont plus difficiles à représenter justement du fait que plusieurs événements puissent avoir lieu pendant une même unité de temps (évé-

nements concurrents). Nous trouvons donc beaucoup moins de travaux sur de tels modèles dans la littérature que pour le temps continu.

Les réseaux de Petri temporisés [62, 100] sont une modélisation à temps discret, mais le comportement de ces modèles n'est pas stochastique. De plus, les techniques d'analyse de ces modèles sont relativement complexes.

Au niveau des réseaux de Petri stochastiques, de nombreux travaux existent, et proposent souvent différentes sémantiques pour traiter le problème des événements concurrents. Lorsque nous considérons un modèle en temps discret, il se peut que plusieurs transitions soient candidates au même instant, comme par exemple les transitions $t1$ et $t2$ dans la figure 1.2a. Les transitions sont dites *concurrentes*. Il faut alors choisir quelle transition effectuer. Nous associons l'événement e à la transition $t1$ et l'événement f à la transition $t2$. La probabilité d'occurrence de e est p, celle de f est q.

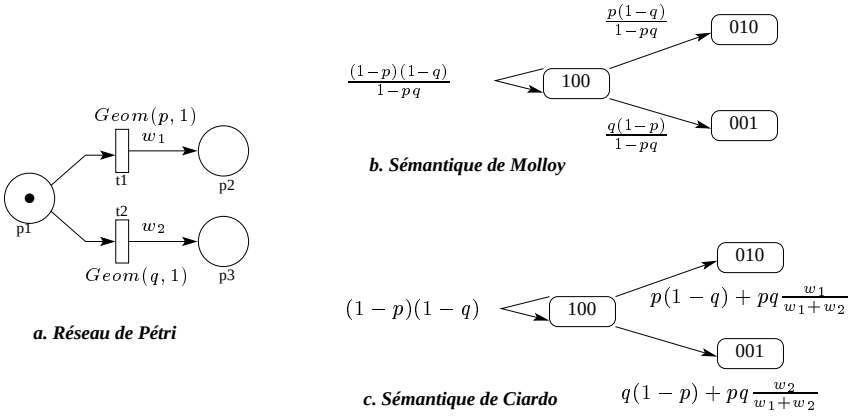


FIG. 1.2 – Différentes sémantiques pour traiter les événements concurrents

Nous trouvons plusieurs sémantiques différentes pour traiter le problème dans la littérature des réseaux de Petri. Dans les premiers travaux de Molloy [77] sur les réseaux de Petri stochastiques à temps discret, les deux événements concurrents e et f ne peuvent pas avoir lieu en même temps. Dans le graphe de marquage du réseau de Petri (figure 1.2b), nous divisons les probabilités par $(1-pq)$ pour avoir une somme des probabilités égale à 1. Ainsi, si aucun événement a lieu (probabilité $(1-p)(1-q)$), nous restons dans le même état, sinon le jeton va dans la place $p2$ ou $p3$.

Dans l'approche de Ciardo [28, 27, 97], nous assignons un poids à chaque événement : $w1$ et $w2$. Les probabilités ne sont alors plus divisées comme pour [77], mais nous considérons que les deux événements peuvent se déclencher simultanément (probabilité pq), et la place du jeton dépend alors des poids (probabilité $w1/(w1+w2)$ que l'événement e ait effectivement lieu). La figure 1.2c illustre cette sémantique.

Des approches pour les réseaux d'automates stochastiques ont également été proposées [2, 54], mais la sémantique n'est pas très explicite en cas de conflit. Pour notre part, nous introduisons la notion de *priorité* associée à chaque événement et déterminons comment

agir en cas de conflit. De plus, un *algorithme de génération de la chaîne de Markov* est proposé, il détecte les événements concurrents et agit en conséquence.

IV.1.2. Agrégation d'états

Une autre forme d'abstraction du système consiste à réduire l'espace d'états afin de simplifier le modèle. En effet, lors de l'analyse de grandes chaînes de Markov, l'utilisation du formalisme SAN n'est pas toujours suffisant pour pallier au problème de l'explosion du nombre d'états.

Pour surmonter ce problème, on cherche à réduire la complexité de la chaîne de Markov étudiée. Heureusement, de nombreux systèmes contiennent un grand nombre de composants identiques. On peut alors exploiter ces réplications de composants pour générer une chaîne de Markov réduite, en effectuant une agrégation exacte [83, 64]. Une telle agrégation est proposée directement sur l'espace d'états de la chaîne de Markov.

Dans le cadre des formalismes de haut niveau structurés, il est intéressant de pratiquer une agrégation directement à partir du modèle de haut niveau, pour éviter de générer la chaîne de Markov et obtenir une représentation compacte de la chaîne de Markov agrégée. Il est alors nécessaire de définir un formalisme de haut niveau qui définit les réplications de composants, et de prouver que le modèle peut être agrégé sous certaines conditions.

Dans le chapitre 3, nous nous plaçons à nouveau dans le cadre des SANs à temps continu, et nous introduisons la notion de **réseaux d'automates stochastiques avec réplication**. Nous tenons compte du fait que, la plupart du temps, un système est constitué de plusieurs entités identiques. Nous définissons donc la notion d'automates répliqués dans un SAN, et nous proposons alors une agrégation exacte sur de tels modèles.

De nombreux travaux ont déjà été effectués sur ce domaine [86, 45] mais aucun à notre connaissance n'a proposé une *agrégation exacte de SANs avec plusieurs répliquas, tenant compte des fonctions*. De plus, nous donnons une *expression tensorielle de la matrice de la chaîne de Markov agrégée*, ce qui permet d'avoir un modèle stocké de façon compacte, et des méthodes de résolution efficaces sur ce modèle.

La deuxième partie de la thèse met l'accent sur les méthodes et algorithmes pour la résolution numérique des modèles SAN.

IV.2. Résolution numérique de modèles SAN

Dans le but de pouvoir analyser les performances de systèmes à grand espace d'états, nous nous sommes également intéressés à l'aspect "méthodes numériques" pour calculer les indices de performances. Comme nous l'avons dit précédemment, la principale difficulté de ces méthodes réside dans la **multiplication d'un vecteur de probabilité par le générateur de la chaîne de Markov**.

En effet, c'est l'opération de base des méthodes itératives utilisées pour calculer des indices de performance, et ces méthodes ont souvent besoin de calculer ce produit un grand

nombre de fois. Les systèmes étudiés sont à grand espace d'états (souvent plus d'un million d'états), il est donc difficile de stocker le vecteur et le générateur, et surtout l'efficacité de l'opération dépend directement de ce stockage.

IV.2.1. ALGORITHMES DE MULTIPLICATION VECTEUR-GÉNÉRATEUR

Le chapitre 4 détaille les différentes approches qui ont été proposées, et notamment les différentes structures de données que nous pouvons utiliser pour stocker les vecteurs de probabilité, le générateur de la chaîne de Markov, et l'espace d'états accessibles du modèle. Le générateur peut être stocké de façon explicite en ne gardant que les éléments non nuls et leur position dans la matrice, comme par exemple en format Harwell-Boeing [84, 90]. Cette approche manque cependant de structure et elle est vite limitée par la mémoire pour les systèmes à grand espace d'états. Parmi les représentations structurées, on remarque notamment les diagrammes de décision [58] et les diagrammes de matrices [30, 31, 75]. Toutes ces approches seront développées ultérieurement.

Pour notre part, nous nous plaçons toujours dans le cadre des réseaux d'automates stochastiques, qui permettent de diminuer les besoins en mémoire par leur approche structurée, basée sur une formule tensorielle. Nous proposons alors une amélioration de l'algorithme classique utilisé pour multiplier un vecteur par un générateur stocké sous forme tensorielle (appelé aussi *descripteur*).

Le nouvel algorithme tient compte du fait que dans de nombreux modèles, *la proportion d'états accessibles est faible*. Ainsi, il utilise uniquement des *structures réduites* (de la taille de l'espace d'états accessibles), ce qui permet un gain en mémoire et en temps d'exécution important lorsque la proportion d'états accessibles est effectivement faible. Nous pouvons alors calculer des indices de performances sur des modèles comportant plus de dix millions d'états en un temps raisonnable.

IV.2.2. MISE EN OEUVRE DANS PEPS ET ANALYSE DES RÉSULTATS

Le chapitre 5 présente le logiciel **PEPS** (*Performance Evaluation of Parallel Systems*) dans sa version 2003. Ces travaux de thèse ont contribué au *développement du logiciel*, avec notamment l'implémentation des nouveaux algorithmes de multiplication, et aussi la participation à la conception d'une nouvelle interface adaptée à la description des réseaux d'automates stochastiques avec répliquas.

Des **résultats numériques** sont alors présentés pour illustrer l'apport des travaux effectués pour l'évaluation des performances des systèmes à grand espace d'états.

Nous comparons notamment dans cette partie les nouveaux algorithmes développés dans cette thèse avec des algorithmes classiques (l'algorithme du shuffle classique, la multiplication par une matrice creuse) sur différents modèles, et nous dressons un ensemble de conclusions pour indiquer quel algorithme il est préférable d'utiliser suivant le modèle étudié. Nous montrons ainsi que les améliorations portées à l'algorithme du shuffle nous permettent d'analyser des systèmes parallèles et distribués avec un espace d'états de plus en

plus grand, et ceci nous semble d'une importance primordiale dans un contexte où de tels systèmes ne font que se complexifier.

Nous montrons également l'intérêt de l'agrégation des SANs avec réplication, qui permet de résoudre en un temps négligeable des modèles qui prenaient beaucoup de temps auparavant. Les techniques d'agrégation utilisées permettent de réduire la taille de l'espace d'état de manière très significative, et la résolution du modèle est alors immédiate.

IV.3. CONCLUSIONS ET PERSPECTIVES

Pour terminer, le chapitre 6 dresse un bilan des travaux effectués, et précise les travaux en cours, ainsi que les perspectives de travaux que nous envisageons pour donner suite à cette thèse.

Un glossaire (page 157) récapitule les définitions des principaux termes couramment employés dans cette thèse, ainsi que des références vers les définitions.

CHAPITRE 2

Les Réseaux d'Automates Stochastiques - SANs

Les réseaux d'automates stochastiques (SANs : Stochastic Automata Networks) [80, 41] permettent de modéliser des systèmes complexes à grand espace d'états, et des techniques de résolution efficaces basées sur une algèbre tensorielle sont alors disponibles. Un automate stochastique est très proche d'une chaîne de Markov, et nous pouvons appliquer dessus les techniques standard de ce domaine. Ainsi, tout système Markovien pourra être facilement représenté. L'intérêt principal des réseaux d'automates stochastiques est qu'ils permettent d'avoir une vision modulaire du système étudié, et de modéliser ensuite les interactions entre les différentes composantes du système.

Nous commencerons par rappeler quelques définitions d'algèbre tensorielle, employées pour exprimer le générateur de la chaîne de Markov associée au réseau d'automates stochastiques. Nous décrirons ensuite les réseaux d'automates stochastiques de façon informelle, afin de comprendre l'outil de modélisation sans préoccupation théorique, puis une description formelle des SANs sera exposée. Enfin, nous présenterons quelques exemples de modélisation, utilisés par la suite pour montrer l'efficacité des algorithmes développés au cours de cette thèse.

I. ALGÈBRE TENSORIELLE

Nous présentons dans cette section succinctement l'algèbre tensorielle classique [34], ainsi qu'une extension, l'algèbre tensorielle généralisée [80, 2, 41]. La différence fondamentale apportée par cette extension est l'introduction du concept d'*éléments fonctionnels*. Une matrice peut désormais être composée d'éléments constants (appartenant à $\mathbb{R}$) ou d'éléments fonctionnels. Un élément fonctionnel est une fonction à valeur dans $\mathbb{R}$, prenant comme arguments les indices de ligne d'une ou de plusieurs matrices.

L'algèbre tensorielle est définie par deux opérateurs matriciels : les *produits tensoriels* (aussi appelés produits de Kronecker) et les *sommes tensorielles*. Nous définissons maintenant ces deux opérateurs, et nous introduisons également la notion de *facteurs normaux*.

I.1. PRODUIT TENSORIEL

I.1.1. PRODUIT TENSORIEL CLASSIQUE

Le produit tensoriel de deux matrices A et B, de dimensions $(\alpha_1 \times \alpha_2)$ et $(\beta_1 \times \beta_2)$ respectivement, est une matrice de dimensions $(\alpha_1\beta_1 \times \alpha_2\beta_2)$. Cette matrice peut être vue comme une matrice avec $\alpha_1 \times \alpha_2$ blocs, chacun avec dimension $\beta_1 \times \beta_2$. La définition de chacun des éléments de la matrice résultante est faite en déterminant à quel bloc l'élément appartient et sa position interne dans le bloc.

✍ **Soit**

A	la matrice nommée A ;
$A \otimes B$	le produit tensoriel des matrices A et B ;
$A \times B$	le produit (traditionnel) des matrices A et B ;
$a_{i,j}$	l'élément dans la ligne i et la colonne j de la matrice A ;
$a_{[i,k],[j,l]}$	l'élément dans la ligne k du i-ème bloc horizontal et la colonne l du j-ème bloc vertical de la matrice A.

Prenons par exemple deux matrices A et B :

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \qquad B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \end{pmatrix}$$

Le produit tensoriel défini par $C = A \otimes B$ est égal à

$$C = \begin{pmatrix} a_{1,1}B & a_{1,2}B \\ a_{2,1}B & a_{2,2}B \end{pmatrix}$$

$$C = \left(\begin{array}{cccc|cccc} a_{1,1}b_{1,1} & a_{1,1}b_{1,2} & a_{1,1}b_{1,3} & a_{1,1}b_{1,4} & a_{1,2}b_{1,1} & a_{1,2}b_{1,2} & a_{1,2}b_{1,3} & a_{1,2}b_{1,4} \\ a_{1,1}b_{2,1} & a_{1,1}b_{2,2} & a_{1,1}b_{2,3} & a_{1,1}b_{2,4} & a_{1,2}b_{2,1} & a_{1,2}b_{2,2} & a_{1,2}b_{2,3} & a_{1,2}b_{2,4} \\ a_{1,1}b_{3,1} & a_{1,1}b_{3,2} & a_{1,1}b_{3,3} & a_{1,1}b_{3,4} & a_{1,2}b_{3,1} & a_{1,2}b_{3,2} & a_{1,2}b_{3,3} & a_{1,2}b_{3,4} \\ \hline a_{2,1}b_{1,1} & a_{2,1}b_{1,2} & a_{2,1}b_{1,3} & a_{2,1}b_{1,4} & a_{2,2}b_{1,1} & a_{2,2}b_{1,2} & a_{2,2}b_{1,3} & a_{2,2}b_{1,4} \\ a_{2,1}b_{2,1} & a_{2,1}b_{2,2} & a_{2,1}b_{2,3} & a_{2,1}b_{2,4} & a_{2,2}b_{2,1} & a_{2,2}b_{2,2} & a_{2,2}b_{2,3} & a_{2,2}b_{2,4} \\ a_{2,1}b_{3,1} & a_{2,1}b_{3,2} & a_{2,1}b_{3,3} & a_{2,1}b_{3,4} & a_{2,2}b_{3,1} & a_{2,2}b_{3,2} & a_{2,2}b_{3,3} & a_{2,2}b_{3,4} \end{array}\right)$$

Dans cet exemple l'élément $c_{4,7}(= a_{2,2}b_{1,3})$ est dans le bloc $(2, 2)$ et sa position interne dans ce bloc est $(1, 3)$. Le produit tensoriel $C = A \otimes B$ est défini algébriquement par l'affectation de la valeur $a_{i,j}b_{k,l}$ à l'élément dans la position (k, l) du bloc (i, j), *i.e.* :

$$c_{[i,k],[j,l]} = a_{i,j}b_{k,l} \qquad \text{avec } i \in [1..\alpha_1],\ j \in [1..\alpha_2],\ k \in [1..\beta_1] \text{ et } l \in [1..\beta_2] \tag{2.1}$$

Cette représentation des éléments d'une matrice correspondant à un produit tensoriel induit un ordre sur les éléments $c_{[i,k],[j,l]}$ qui est l'ordre lexicographique sur les doublets d'indice.

I.1.2. FACTEURS NORMAUX

Un cas spécifique de produit tensoriel est le produit tensoriel d'une matrice carrée par une matrice identité. Appelons ces produits des *facteurs normaux*. Avec une matrice carrée A et une matrice identité I_n de dimension n, deux facteurs normaux sont possibles : $(A \otimes I_n)$ et $(I_n \otimes A)$.

Reprenons la matrice A de l'exemple précédent et une matrice identité de taille 3 :

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \qquad I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Le facteur normal $A \otimes I_3$ est égal à

$$\left(\begin{array}{ccc|ccc} a_{1,1} & 0 & 0 & a_{1,2} & 0 & 0 \\ 0 & a_{1,1} & 0 & 0 & a_{1,2} & 0 \\ 0 & 0 & a_{1,1} & 0 & 0 & a_{1,2} \\ \hline a_{2,1} & 0 & 0 & a_{2,2} & 0 & 0 \\ 0 & a_{2,1} & 0 & 0 & a_{2,2} & 0 \\ 0 & 0 & a_{2,1} & 0 & 0 & a_{2,2} \end{array}\right)$$

Le facteur normal $I_3 \otimes A$ est égal à

$$\left(\begin{array}{cc|cc|cc} a_{1,1} & a_{1,2} & 0 & 0 & 0 & 0 \\ a_{2,1} & a_{2,2} & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & a_{1,1} & a_{1,2} & 0 & 0 \\ 0 & 0 & a_{2,1} & a_{2,2} & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & a_{1,1} & a_{1,2} \\ 0 & 0 & 0 & 0 & a_{2,1} & a_{2,2} \end{array}\right)$$

Un cas encore plus particulier de produit tensoriel est le produit de deux matrices identités. Ce produit est une matrice identité dont la dimension est égale au produit des dimensions de chacune des matrices, *i.e.* :

$$I_n \otimes I_m = I_{nm}$$

I.1.3. PRODUIT TENSORIEL GÉNÉRALISÉ

Nous considérons maintenant des matrices pouvant posséder des éléments fonctionnels, c'est-à-dire des fonctions à valeur dans $\mathbb{R}$, prenant comme arguments les indices de ligne d'une ou de plusieurs matrices.

Un élément fonctionnel qui possède l'indice de ligne d'une matrice A comme argument est dit dépendant de la matrice A. Par abus de langage, nous appelons arguments d'un élément fonctionnel toutes les matrices dont l'élément est dépendant. Une matrice contenant au moins un élément fonctionnel dépendant de la matrice A est dite *dépendante* de la matrice A. Les *arguments* d'une matrice sont l'union des arguments de tous ses éléments.

> *Par hypothèse, toutes les matrices ont leurs indices de ligne de 1 à n, n étant la dimension de la matrice.*

✍ **Soit**

$(a_1 \,..\, a_n)$ un espace associé à l'ensemble des lignes $[1..n]$ de la matrice A (de dimension n). A chaque ligne correspond un élément, qui servira à évaluer les fonctions par la suite.

a_k l'élément correspondant à l'indice de la ligne k de la matrice A ;

$A(\mathcal{B},\mathcal{C})$ la matrice fonctionnelle A qui possède comme arguments les matrices B et C ;

$a_{i,j}(\mathcal{B},\mathcal{C})$ l'élément fonctionnel i,j de la matrice $A(\mathcal{B},\mathcal{C})$;

$A(b_k,\mathcal{C})$ la matrice fonctionnelle $A(\mathcal{B},\mathcal{C})$ où l'indice de ligne de la matrice B est déjà connu et égal à k (cette matrice peut être considéré comme dépendante de la matrice C seulement) ;

$a_{i,j}(b_k,\mathcal{C})$ l'élément fonctionnel i,j de la matrice $A(b_k,\mathcal{C})$;

$A(b_k,c_l)$ la matrice fonctionnelle $A(\mathcal{B},\mathcal{C})$ où les indices de ligne des matrices B et C sont déjà connus et égaux respectivement à k et l (cette matrice, ayant tous ses arguments connus, est considérée comme constante) ;

$a_{i,j}(b_k,c_l)$ l'élément constant (élément fonctionnel évalué) i,j de la matrice $A(b_k,c_l)$;

$A(\mathcal{B}) \underset{g}{\otimes} B(\mathcal{A})$ le produit tensoriel généralisé entre les matrices $A(\mathcal{B})$ et $B(\mathcal{A})$.

Prenons par exemple deux matrices $A(\mathcal{B})$ et $B(\mathcal{A})$ définies par :

$$A(\mathcal{B}) = \begin{pmatrix} a_{1,1}[\mathcal{B}] & a_{1,2}[\mathcal{B}] \\ a_{2,1}[\mathcal{B}] & a_{2,2}[\mathcal{B}] \end{pmatrix} \qquad B(\mathcal{A}) = \begin{pmatrix} b_{1,1}[\mathcal{A}] & b_{1,2}[\mathcal{A}] & b_{1,3}[\mathcal{A}] \\ b_{2,1}[\mathcal{A}] & b_{2,2}[\mathcal{A}] & b_{2,3}[\mathcal{A}] \\ b_{3,1}[\mathcal{A}] & b_{3,2}[\mathcal{A}] & b_{3,3}[\mathcal{A}] \end{pmatrix}$$

Le produit tensoriel défini par $C = A(\mathcal{B}) \underset{g}{\otimes} B(\mathcal{A})$ est égal à

$$C = \left(\begin{array}{ccc|ccc} a_{1,1}(b_1)b_{1,1}(a_1) & a_{1,1}(b_1)b_{1,2}(a_1) & a_{1,1}(b_1)b_{1,3}(a_1) & a_{1,2}(b_1)b_{1,1}(a_1) & a_{1,2}(b_1)b_{1,2}(a_1) & a_{1,2}(b_1)b_{1,3}(a_1) \\ a_{1,1}(b_2)b_{2,1}(a_1) & a_{1,1}(b_2)b_{2,2}(a_1) & a_{1,1}(b_2)b_{2,3}(a_1) & a_{1,2}(b_2)b_{2,1}(a_1) & a_{1,2}(b_2)b_{2,2}(a_1) & a_{1,2}(b_2)b_{2,3}(a_1) \\ a_{1,1}(b_3)b_{3,1}(a_1) & a_{1,1}(b_3)b_{3,2}(a_1) & a_{1,1}(b_3)b_{3,3}(a_1) & a_{1,2}(b_3)b_{3,1}(a_1) & a_{1,2}(b_3)b_{3,2}(a_1) & a_{1,2}(b_3)b_{3,3}(a_1) \\ \hline a_{2,1}(b_1)b_{1,1}(a_2) & a_{2,1}(b_1)b_{1,2}(a_2) & a_{2,1}(b_1)b_{1,3}(a_2) & a_{2,2}(b_1)b_{1,1}(a_2) & a_{2,2}(b_1)b_{1,2}(a_2) & a_{2,2}(b_1)b_{1,3}(a_2) \\ a_{2,1}(b_2)b_{2,1}(a_2) & a_{2,1}(b_2)b_{2,2}(a_2) & a_{2,1}(b_2)b_{2,3}(a_2) & a_{2,2}(b_2)b_{2,1}(a_2) & a_{2,2}(b_2)b_{2,2}(a_2) & a_{2,2}(b_2)b_{2,3}(a_2) \\ a_{2,1}(b_3)b_{3,1}(a_2) & a_{2,1}(b_3)b_{3,2}(a_2) & a_{2,1}(b_3)b_{3,3}(a_2) & a_{2,2}(b_3)b_{3,1}(a_2) & a_{2,2}(b_3)b_{3,2}(a_2) & a_{2,2}(b_3)b_{3,3}(a_2) \end{array}\right)$$

La définition algébrique du produit tensoriel généralisé $C = A(\mathcal{B}) \underset{g}{\otimes} B(\mathcal{A})$ est faite par l'affectation de la valeur $a_{i,j}(b_k)b_{k,l}(a_i)$ à l'élément $c_{[i,k],[j,l]}$, *i.e.* :

$$c_{[i,k],[j,l]} = a_{i,j}(b_k)b_{k,l}(a_i) \quad \text{avec } i,j \in [1..n_A] \text{ et } k,l \in [1..n_B] \tag{2.2}$$

I.2. SOMME TENSORIELLE

I.2.1. SOMME TENSORIELLE CLASSIQUE

✍ **Soit**

$A \oplus B$	la somme tensorielle des matrices carrées[1] A et B ;
$A + B$	la somme (traditionnelle) des matrices A et B ;
n_A	la dimension (nombre de lignes et de colonnes) de la matrice carrée A ;
I_n	la matrice identité de dimension n.

La somme tensorielle de deux matrices carrées A et B est définie comme la somme de facteurs normaux de chacune des matrices selon la formule :

$$A \oplus B = (A \otimes I_{n_B}) + (I_{n_A} \otimes B) \tag{2.3}$$

Prenons par exemple les matrices A et B définies par :

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \qquad B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}$$

La somme tensorielle définie par $C = A \oplus B$ est égale à

$$C = \left(\begin{array}{ccc|ccc} a_{1,1} & 0 & 0 & a_{1,2} & 0 & 0 \\ 0 & a_{1,1} & 0 & 0 & a_{1,2} & 0 \\ 0 & 0 & a_{1,1} & 0 & 0 & a_{1,2} \\ \hline a_{2,1} & 0 & 0 & a_{2,2} & 0 & 0 \\ 0 & a_{2,1} & 0 & 0 & a_{2,2} & 0 \\ 0 & 0 & a_{2,1} & 0 & 0 & a_{2,2} \end{array}\right) + \left(\begin{array}{ccc|ccc} b_{1,1} & b_{1,2} & b_{1,3} & 0 & 0 & 0 \\ b_{2,1} & b_{2,2} & b_{2,3} & 0 & 0 & 0 \\ b_{3,1} & b_{3,2} & b_{3,3} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & b_{1,1} & b_{1,2} & b_{1,3} \\ 0 & 0 & 0 & b_{2,1} & b_{2,2} & b_{2,3} \\ 0 & 0 & 0 & b_{3,1} & b_{3,2} & b_{3,3} \end{array}\right)$$

$$C = \left(\begin{array}{ccc|ccc} a_{1,1}+b_{1,1} & b_{1,2} & b_{1,3} & a_{1,2} & 0 & 0 \\ b_{2,1} & a_{1,1}+b_{2,2} & b_{2,3} & 0 & a_{1,2} & 0 \\ b_{3,1} & b_{3,2} & a_{1,1}+b_{3,3} & 0 & 0 & a_{1,2} \\ \hline a_{2,1} & 0 & 0 & a_{2,2}+b_{1,1} & b_{1,2} & b_{1,3} \\ 0 & a_{2,1} & 0 & b_{2,1} & a_{2,2}+b_{2,2} & b_{2,3} \\ 0 & 0 & a_{2,1} & b_{3,1} & b_{3,2} & a_{2,2}+b_{3,3} \end{array}\right)$$

[1] Bien que le produit tensoriel soit défini pour des matrices non carrées, la somme tensorielle est définie exclusivement pour les matrices carrées.

La somme tensorielle $C = A \oplus B$ est définie algébriquement par l'affectation de la valeur $a_{i,j}\delta_{k,l} + \delta_{i,j}b_{k,l}$ à l'élément dans la position (k,l) du bloc (i,j), *i.e.* :

$$c_{[i,k],[j,l]} = a_{i,j}\delta_{k,l} + \delta_{i,j}b_{k,l} \quad \text{avec } i,j \in [1..n_A] \text{ et } k,l \in [1..n_B] \tag{2.4}$$

I.2.2. SOMME TENSORIELLE GÉNÉRALISÉE

La définition de la somme tensorielle généralisée est faite en utilisant des produits tensoriels généralisés sur l'équation (2.3) :

$$A \underset{g}{\oplus} B = (A \underset{g}{\otimes} I_{n_B}) + (I_{n_A} \underset{g}{\otimes} B) \tag{2.5}$$

Reprenons les matrices $A(\mathcal{B})$ et $B(\mathcal{A})$ utilisées pour décrire le produit tensoriel généralisé (section I.1.3, page 25). La somme tensorielle définie par $C = A(\mathcal{B}) \underset{g}{\oplus} B(\mathcal{A})$ est égale à :

$$C = \left(\begin{array}{ccc|ccc} a_{1,1}(b_1)+b_{1,1}(a_1) & b_{1,2}(a_1) & b_{1,3}(a_1) & a_{1,2}(b_1) & 0 & 0 \\ b_{2,1}(a_1) & a_{1,1}(b_2)+b_{2,2}(a_1) & b_{2,3}(a_1) & 0 & a_{1,2}(b_2) & 0 \\ b_{3,1}(a_1) & b_{3,2}(a_1) & a_{1,1}(b_3)+b_{3,3}(a_1) & 0 & 0 & a_{1,2}(b_3) \\ \hline a_{2,1}(b_1) & 0 & 0 & a_{2,2}(b_1)+b_{1,1}(a_2) & b_{1,2}(a_2) & b_{1,3}(a_2) \\ 0 & a_{2,1}(b_2) & 0 & b_{2,1}(a_2) & a_{2,2}(b_2)+b_{2,2}(a_2) & b_{2,3}(a_2) \\ 0 & 0 & a_{2,1}(b_3) & b_{3,1}(a_2) & b_{3,2}(a_2) & a_{2,2}(b_3)+b_{3,3}(a_2) \end{array}\right)$$

La définition algébrique de la somme tensorielle généralisée $C = A(\mathcal{B}) \underset{g}{\oplus} B(\mathcal{A})$ est faite par l'affectation de la valeur $a_{i,j}(b_k)\delta_{k,l} + b_{k,l}(a_i)\delta_{i,j}$ à l'élément $c_{[i,k],[j,l]}$, *i.e.* :

$$c_{[i,k],[j,l]} = a_{i,j}(b_k)\delta_{k,l} \; + \; b_{k,l}(a_i)\delta_{i,j} \quad \text{avec } i,j \in [1..n_A] \text{ et } k,l \in [1..n_B] \tag{2.6}$$

II. PRÉSENTATION INFORMELLE DES SANS

Les bases d'algèbre tensorielle ayant été rappelées, nous nous intéressons maintenant au formalisme des réseaux d'automates stochastiques, et commençons par en faire une présentation informelle.

L'approche modulaire des réseaux d'automates stochastiques nous amène à décrire le système comme un ensemble de sous-systèmes qui interagissent entre eux. Chaque sous-système est modélisé par un automate stochastique, et des règles établies entre les états internes de chaque automate permettent de décrire les interactions entre les sous-systèmes. Les réseaux d'automates stochastiques peuvent être utilisés pour modéliser des systèmes à échelle de temps continu ou discret.

Nous présenterons tout d'abord la notion d'automates et d'événements, qui sont à la base des SANs. Nous verrons ensuite comment, à partir d'un SAN, nous pouvons obtenir une chaîne de Markov équivalente. Enfin, nous introduirons le concept de descripteur Markovien, qui se base sur l'algèbre tensorielle.

II.1. AUTOMATES ET ÉVÉNEMENTS

Un réseau d'automates stochastiques est constitué d'un ensemble d'automates et d'un ensemble d'événements. Les **états internes**, ou **états locaux** d'un automate correspondent aux différents états possibles du sous-système modélisé par cet automate.

L'occurrence d'un événement peut modifier l'état interne d'un ou de plusieurs automates. Lorsqu'un seul automate est impliqué, l'événement est dit **local**. Sinon, nous parlons d'événement **synchronisant**. Dans tous les cas, nous avons un changement de l'**état global** du système, qui est une combinaison de tous les états internes de chaque automate.

Chaque automate est ainsi caractérisé par un ensemble d'états et un ensemble de **transitions**, étiquetées par des événements, qui permettent de changer l'état de l'automate.

Pour représenter graphiquement un réseau d'automates stochastiques, nous pouvons ainsi associer un graphe à chaque automate. Les **noeuds** du graphe correspondent aux états locaux de l'automate, et les **arcs** sont étiquetés par une liste d'événements. Il se peut en effet que plusieurs événements distincts produisent le même changement d'état local dans un automate. Dans ce cas, nous avons formellement une transition par événement, et l'arc du graphe représente toutes ces transitions en même temps.

Nous allons présenter ces différentes notions à travers un exemple introductif, l'exclusion mutuelle (échelle de temps continu).

II.1.1. LE PROBLÈME DE L'EXCLUSION MUTUELLE

Deux clients désirent utiliser une unique ressource. Chaque client peut être soit dans l'état *repos* (il n'utilise pas la ressource), soit dans l'état *activité* (il utilise la ressource). Nous modélisons donc chaque client par un automate à deux états : $\mathcal{A}^{(1)}$ et $\mathcal{A}^{(2)}$ sont les automates décrivant respectivement les clients 1 et 2. Les états internes de chacun de ces automates sont *repos* et *activité*.

Les différents événements pouvant avoir lieu sont :

- la prise de la ressource par le premier client, p_1 ;
- la prise de la ressource par le deuxième client, p_2 ;
- la relâche de la ressource par le premier client, r_1 ;
- la relâche de la ressource par le deuxième client, r_2.

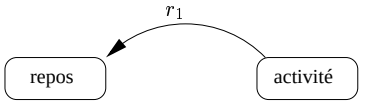
La durée de résidence dans l'état *repos* est pour chaque client une variable aléatoire exponentielle de taux λ. Nous disons que λ est le **taux de franchissement** associé aux événements p_1 et p_2. De la même façon, le taux de franchissement de r_1 et de r_2 est μ.

Pour assurer l'exclusion mutuelle, nous devons modéliser les interactions entre les deux

clients, pour exprimer notamment le fait que les deux clients ne peuvent pas être simultanément dans l'état *activité*. Nous allons voir comment décrire ces interactions, d'une part en n'utilisant que des événements locaux, et d'autre part à l'aide d'événements synchronisant.

II.1.2. LES ÉVÉNEMENTS LOCAUX

L'occurrence d'un événement local correspond au changement de l'état interne d'un seul automate. Nous associons à chaque événement local un taux de franchissement. Dans notre exemple, le premier client passe de l'état *activité* à l'état *repos* lors de l'occurrence de r_1, avec un taux μ. L'occurrence de cet événement de dépend pas du reste du système, et ne modifie que l'état interne de l'automate correspondant au premier client. La figure 2.1 montre la portion de l'automate correspondante.



Automate du premier client, $\mathcal{A}^{(1)}$

FIG. 2.1 – Exemple d'événement local

Nous pouvons également modéliser la prise de ressource à l'aide d'un événement local p_1, mais il faut exprimer le fait que la ressource ne peut être prise que si elle est libre. Pour cela, nous pouvons utiliser des taux de franchissement **fonctionnels**. Dans ce cas, le taux de franchissement est variable, il dépend de l'état des autres automates. C'est une fonction de l'état global. Dans notre exemple, l'état global est une combinaison des états des deux automates $\mathcal{A}^{(1)}$ et $\mathcal{A}^{(2)}$. Ainsi, si les deux clients sont dans l'état *repos*, l'état global est :

(état($\mathcal{A}^{(1)}$) = *repos*, état($\mathcal{A}^{(2)}$) = *repos*).

Dans notre exemple, nous introduisons une fonction f qui exprime le fait que le client peut prendre la ressource uniquement si elle n'est pas encore occupée[2] :

f(état global) = δ(état($\mathcal{A}^{(1)}$) = *repos* et état($\mathcal{A}^{(2)}$) = *repos*)

Le taux de franchissement associé aux événements p_1 et p_2 est alors λf, c'est une fonction de l'état global du système. Nous pouvons représenter notre système par le réseau d'automates stochastiques de la figure 2.2, où le taux de franchissement associé à r_1 et r_2 reste μ.

II.1.3. LES ÉVÉNEMENTS SYNCHRONISANT

Il est possible de décrire le même système, en modélisant les interactions entre les deux clients de façon différente. Pour cela, nous introduisons un automate supplémentaire $\mathcal{A}^{(3)}$ qui modélise l'état de la ressource. Cet automate a deux états internes *libre* et *occupée*. Nous synchronisons alors les clients avec la ressource :

– une prise de ressource doit être accompagnée du passage de l'état de la ressource de l'état *libre* vers *occupée* ;

[2] Rappelons que $\delta(b)$ est une fonction qui vaut 1 si l'expression b est vrai, sinon cette fonction vaut 0.

FIG. 2.2 – L'exclusion mutuelle, événements locaux

– une relâche de ressource doit être accompagnée du passage de l'état de la ressource de l'état *occupée* vers *libre*.

Ainsi, nous synchronisons les clients avec la ressource, grâce aux événements p_1, p_2, r_1, et r_2. Un événement qui synchronise plusieurs automates ne peut avoir lieu que si chaque automate impliqué par la synchronisation est dans un état où la synchronisation est possible (un arc étiqueté par l'événement synchronisant part de cet état).

Dans notre exemple les événements p_1 et r_1 synchronisent $\mathcal{A}^{(1)}$ et $\mathcal{A}^{(3)}$, et ils ont des taux de franchissement λ et μ respectivement. Les événements p_2 et r_2 synchronisent $\mathcal{A}^{(2)}$ et $\mathcal{A}^{(3)}$, et ils ont des taux de franchissement λ et μ respectivement.

La figure 2.3 décrit ce modèle. Ainsi, l'événement de prise de la ressource par le premier client p_1 ne peut avoir lieu que si l'état de $\mathcal{A}^{(1)}$ est *repos* et si l'état de $\mathcal{A}^{(3)}$ est *libre*. Dans ce cas, les deux transitions des automates, qui correspondent aux arcs du graphe étiquetés par p_1, peuvent avoir lieu simultanément, avec un taux de franchissement λ.

Noter également que lorsqu'un arc est étiqueté par une liste d'événements (séparés par des virgules), plusieurs transitions peuvent avoir lieu lors de l'occurrence de l'un de ces événements, quel qu'il soit. Formellement, nous définissons une transition distincte par événement de la liste.

Les taux de franchissement associés à des événements synchronisants peuvent bien entendu être fonctionnels, tout comme ceux des événements locaux.

II.1.4. PROBABILITÉ DE ROUTAGE

Il se peut qu'un même événement, en partant du même état, puisse mener dans plusieurs états différents. Dans ce cas, il faut définir la probabilité d'aller dans chacun des états d'arrivée, et la somme de ces probabilités doit être égale à 1. Ces probabilités sont appelées **probabilités de routage**, et elles ne sont pas associées un événement mais à chaque transition.

Par défaut, lorsqu'il y a un seul état d'arrivée pour un événement en partant d'un état donné, la probabilité de routage est 1, et elle n'est pas marquée dans le SAN (cf exemples précédents).

En revanche, lorsque plusieurs états d'arrivée sont possibles, on rajoute la probabilité de routage sur les étiquettes de l'arc correspondant à la transition et à l'événement, en la mettant entre parenthèse derrière l'étiquette de l'événement : événement(probabilité de

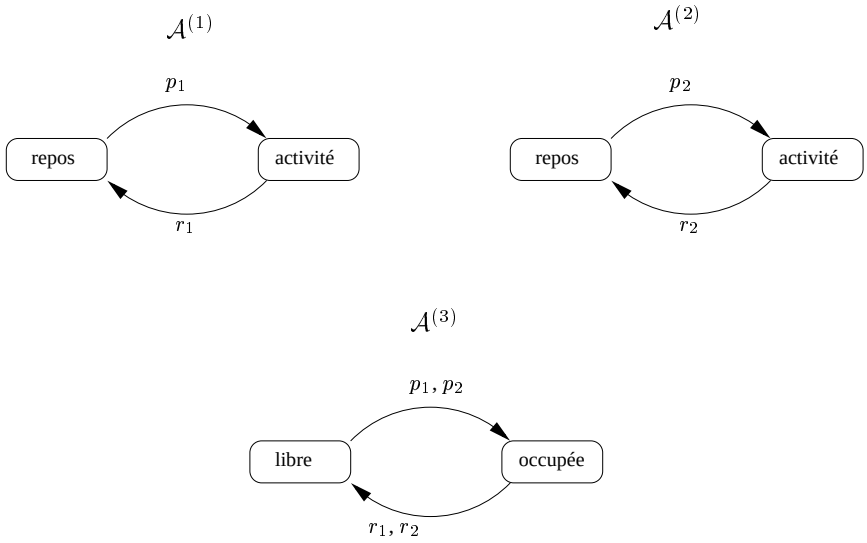


FIG. 2.3 – L'exclusion mutuelle, événements synchronisant

routage).

La figure 2.4 présente un exemple d'étude complet avec des événements locaux et synchronisant, des fonctions, et du routage. Le premier automate $\mathcal{A}^{(1)}$ possède trois états $S0$, $S1$, $S2$; le deuxième automate $\mathcal{A}^{(2)}$ possède deux états $S0$ et $S1$. Les événements sont :

- loc_1, loc_2 et loc_3, événements locaux de $\mathcal{A}^{(1)}$, avec des taux respectivement égaux à λ_1, λ_2 et λ_3 ;
- evt_s, événement synchronisant qui concerne $\mathcal{A}^{(1)}$ et $\mathcal{A}^{(2)}$, avec un taux λ_s ;
- evt_f, événement local de $\mathcal{A}^{(2)}$, avec un taux fonctionnel f :

$$f = \begin{cases} \mu_1 & \text{si } \mathcal{A}^{(1)} \text{ est dans l'état } S0; \\ 0 & \text{si } \mathcal{A}^{(1)} \text{ est dans l'état } S1; \\ \mu_2 & \text{si } \mathcal{A}^{(1)} \text{ est dans l'état } S2. \end{cases}$$

Lorsque $\mathcal{A}^{(1)}$ est dans l'état $S2$ et $\mathcal{A}^{(2)}$ est dans l'état $S1$, l'événement evt_s peut avoir lieu avec le taux λ_s, et le nouvel état de $\mathcal{A}^{(1)}$ est soit $S1$ (probabilité π), soit $S0$ (probabilité $1 - \pi$). Dans tous les cas, le nouvel état de $\mathcal{A}^{(2)}$ est $S0$.

II.2. AUTOMATE ÉQUIVALENT

A partir d'un réseau d'automates stochastiques, il est possible de construire un automate équivalent. Cet automate est le graphe d'une chaîne de Markov décrivant le système.

Nous verrons tout d'abord comment construire cet automate sur un exemple, puis nous détaillerons les propriétés de Markov de cet automate. Enfin, nous motiverons notre intérêt

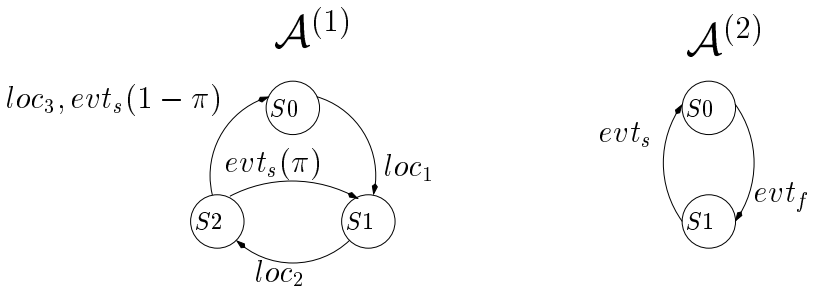


FIG. 2.4 – Exemple avec routage

pour les réseaux d'automates stochastiques dans une petite conclusion.

II.2.1. CONSTRUCTION DE L'AUTOMATE ÉQUIVALENT

Les états de l'automate équivalent correspondent aux états possibles du système que nous modélisons. Il s'agit donc des états globaux du réseau d'automate stochastique (cf section II.1). Les arcs de cet automate sont étiquetés par les taux de franchissement des événements lorsqu'ils changent l'état du système.

L'automate équivalent au réseau d'automates stochastiques de la figure 2.4 est présenté dans la figure 2.5.

Un état global du système est alors un couple constitué d'un état de $\mathcal{A}^{(1)}$ et d'un état de $\mathcal{A}^{(2)}$. Nous notons Sij l'état (Si, Sj), où $i \in 0, 1, 2$ et $j \in 0, 1$ (Si est l'état de $\mathcal{A}^{(1)}$, et Sj l'état de $\mathcal{A}^{(2)}$).

A un événement local de l'automate $\mathcal{A}^{(1)}$, permettant de passer de l'état Si_1 à l'état Si_2, nous associons dans l'automate équivalent les transitions de l'état Si_1j à l'état Si_2j, avec $j \in 0, 1$, étiquetées par le taux de franchissement de l'événement. Nous connaissons alors l'état global du système, les fonctions peuvent être évaluées.

Pour les événements synchronisant, nous rajoutons un arc effectuant le changement d'état de chaque automate concerné par la synchronisation, étiqueté par le taux de franchissement de l'événement.

II.2.2. CHAÎNE DE MARKOV

Considérons le graphe états-transitions qui représente l'automate équivalent. Les transitions sont étiquetées par des variables aléatoires indépendantes, de distribution exponentielle. Ainsi, à un instant donné, l'état futur ne dépend que de l'état présent. De plus, le nombre d'états est fini.

Ce graphe est donc le graphe d'une chaîne de Markov.

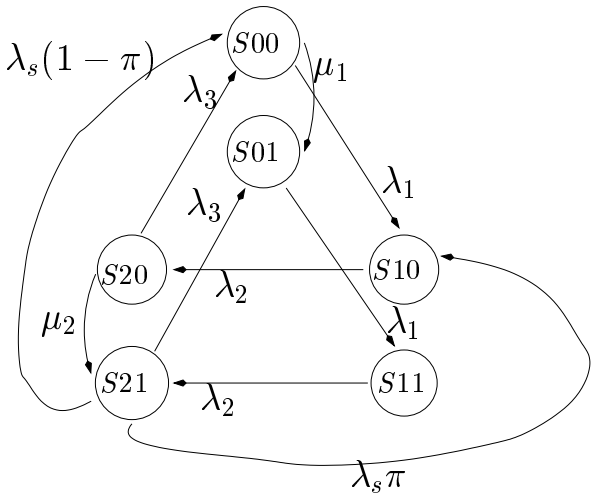


FIG. 2.5 – Automate équivalent - Exemple avec routage

II.2.3. CONCLUSION

L'automate équivalent représente le système de façon non modulaire, son interprétation est plus délicate à première vue. Cependant, les concepts utilisés sont plus simples, nous avons une chaîne de Markov standard avec des taux de franchissement associés aux transitions. Etant donné que le système est constitué de 6 états, le générateur de la chaîne de Markov est une matrice de taille $6^2 = 36$.

La représentation du même système en tant que réseaux d'automates stochastiques permet d'avoir une vision modulaire du système. Dans cette représentation, les interactions entre les différents sous-systèmes sont modélisées par des taux de franchissement fonctionnels et des événements synchronisant. Chaque automate $\mathcal{A}^{(i)}$ est alors décrit par un ensemble de matrices de taille n_i^2, n_i étant le nombre d'états de l'automate. Le générateur de la chaîne de Markov peut s'exprimer sous forme tensorielle à partir des différentes matrices, nous l'appelons le **descripteur**. Nous allons maintenant introduire le concept de formule tensorielle et de descripteur en nous basant sur les exemples précédents.

II.3. DESCRIPTEUR MARKOVIEN

L'un des avantages apporté par le formalisme des réseaux d'automates stochastiques est la facilité de décrire le générateur de la chaîne de Markov associée au modèle complet de façon compacte, grâce à une formule mathématique appelée descripteur [81, 41]. Des techniques similaires sont proposées pour d'autres formalismes, notamment pour les réseaux de Pétri stochastiques généralisés [38, 39], les algèbres de processus [60], ...

Si nous considérons un réseau d'automates stochastiques sans événement synchronisant, nous associons une matrice de transition locale $Q_l^{(i)}$ à chacun des automates $\mathcal{A}^{(i)}$. Si $x^{(i)}$ et $y^{(i)}$ sont deux états locaux distincts de l'automate $\mathcal{A}^{(i)}$, l'élément en ligne $x^{(i)}$ et colonne $y^{(i)}$ de la matrice (les états peuvent être considérés comme étant dans $[1..n_i]$, où n_i est le nombre d'états locaux de $\mathcal{A}^{(i)}$) représente la somme des taux de franchissement des événements locaux permettant de passer de $x^{(i)}$ à $y^{(i)}$ dans l'automate concerné. Le générateur de l'automate équivalent à ce modèle Q est alors égal à une somme tensorielle[3] des matrices de transition locales.

Les taux de franchissement fonctionnels n'apportent pas de modifications structurelles au descripteur, mais l'utilisation d'algèbre tensorielle généralisée[4] est alors nécessaire. En effet, *l'interprétation des matrices de transition permet de faire une correspondance biunivoque entre l'état d'un automate dans un SAN et l'indice de ligne d'une matrice*. Les taux de franchissement fonctionnels correspondent donc à la définition des éléments fonctionnels donnée dans la section I.

Si nous reprenons l'exemple de SAN décrit par la figure 2.2, nous obtenons le descripteur suivant :

$$Q = Q_l^{(1)} \underset{g}{\oplus} Q_l^{(2)} = \begin{pmatrix} -\lambda f & \lambda f \\ \mu & -\mu \end{pmatrix} \bigoplus_g \begin{pmatrix} -\lambda f & \lambda f \\ \mu & -\mu \end{pmatrix}$$

$$Q = \left(\begin{array}{cc|cc} -2\lambda & \lambda & \lambda & 0 \\ \mu & -\mu & 0 & 0 \\ \hline \mu & 0 & -\mu & 0 \\ 0 & \mu & \mu & -2\mu \end{array}\right)$$

La présence d'événements synchronisants nous amène à associer plusieurs matrices à chaque automate $\mathcal{A}^{(i)}$. Déjà, une matrice de transition *locale* $Q_l^{(i)}$ regroupe tous les taux des événements locaux de l'automate. Ensuite, pour chaque événement synchronisant e du modèle, nous associons à l'automate une paire de matrices de transition *synchronisante*. $Q_{e^+}^{(i)}$ décrit l'occurrence de l'événement synchronisant e (taux positifs), et $Q_{e^-}^{(i)}$ fait l'ajustement diagonal correspondant aux taux exprimés dans la première matrice (taux négatifs).

Si l'automate $\mathcal{A}^{(i)}$ n'est pas concerné par l'événement e, alors $Q_{e^+}^{(i)} = Q_{e^-}^{(i)} = I_{n_i}$, où n_i est le nombre d'états de l'automate, et I_n est la matrice identité de dimension n. En effet, cet automate ne changera pas d'état lors de l'occurrence de l'événement e.

Pour chaque événement synchronisant, un des automates est choisi comme automate **maître**. Ce choix peut correspondre à une interprétation du système que nous modélisons, ou bien il peut s'agir d'un choix arbitraire. Ce choix n'apporte aucune restriction à la généralité du formalisme, il sert uniquement à mieux comprendre la structure du descripteur. Les autres automates concernés par l'événement sont appelés automates **esclaves**.

La matrice $Q_{e^+}^{(i)}$ correspondant à l'automate maître de l'événement e contient le taux de franchissement de e. Pour les matrices des automates esclaves, le taux est égal à 1. De

[3] La définition de la somme tensorielle se trouve dans la section I.2, page 27.

[4] La définition de l'algèbre tensorielle généralisée se trouve dans la section I, page 23.

façon analogue, la matrice $Q_{e^-}^{(i)}$ contient des taux d'ajustement négatifs uniquement pour l'automate maître. Les taux dans les matrices d'ajustement associées aux automates esclaves sont égaux à 1, à l'image des matrices d'occurrence. L'existence de probabilités de routage conduit à multiplier les taux par ces probabilités dans les matrices.

Le descripteur se décompose donc en deux parties : une partie dite *locale*, qui correspond aux événements locaux (Q_l), et une partie dite *synchronisante*, qui regroupe tous les événements synchronisant. La partie locale est définie comme une somme tensorielle des matrices locales de chaque automate. Pour la partie synchronisante, nous effectuons pour chaque événement le produit tensoriel des matrices d'occurrence et le produit tensoriel des matrices d'ajustement. La partie synchronisante est la somme de ces produits tensoriels pour tous les événements synchronisants. L'utilisation de l'algèbre tensorielle généralisée est nécessaire dès lors qu'il y a des taux de franchissement fonctionnels.

Nous illustrons ce concept de descripteur sur l'exemple de la figure 2.4. Nous avons choisi arbitrairement que l'automate $\mathcal{A}^{(1)}$ était le maître de l'événement synchronisant evt_s, mais le choix du deuxième automate donne exactement les mêmes matrices $Q_{evt_s^+}$ et $Q_{evt_s^+}$, et donc un générateur absolument identique.

Partie locale :

$$Q_l = Q_l^{(1)} \underset{g}{\oplus} Q_l^{(2)} = \begin{pmatrix} -\lambda_1 & \lambda_1 & 0 \\ 0 & -\lambda_2 & \lambda_2 \\ \lambda_3 & 0 & -\lambda_3 \end{pmatrix} \bigoplus_g \begin{pmatrix} -f & f \\ 0 & 0 \end{pmatrix}$$

$$Q_l = \left(\begin{array}{cc|cc|cc} -(\lambda_1+\mu_1) & \mu_1 & \lambda_1 & 0 & 0 & 0 \\ 0 & -\lambda_1 & 0 & \lambda_1 & 0 & 0 \\ \hline 0 & 0 & -\lambda_2 & 0 & \lambda_2 & 0 \\ 0 & 0 & 0 & -\lambda_2 & 0 & \lambda_2 \\ \hline \lambda_3 & 0 & 0 & 0 & -(\lambda_3+\mu_2) & \mu_2 \\ 0 & \lambda_3 & 0 & 0 & 0 & -\lambda_3 \end{array}\right)$$

Partie synchronisante positive :

$$Q_{evt_s^+} = Q_{evt_s^+}^{(1)} \underset{g}{\otimes} Q_{evt_s^+}^{(2)} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \lambda_s(1-\pi) & \lambda_s\pi & 0 \end{pmatrix} \bigotimes_g \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

$$Q_{evt_s^+} = \left(\begin{array}{cc|cc|cc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ \lambda_s(1-\pi) & 0 & \lambda_s\pi & 0 & 0 & 0 \end{array}\right)$$

Partie synchronisante négative :

$$Q_{evt_s^-} = Q_{evt_s^-}^{(1)} \underset{g}{\otimes} Q_{evt_s^-}^{(2)} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -\lambda_s \end{pmatrix} \bigotimes_g \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

$$Q_{evt_s^-} = \left(\begin{array}{cc|cc|cc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\lambda_s \end{array}\right)$$

Générateur de l'automate global :

$$Q = Q_l + Q_{evt_s^+} + Q_{evt_s^-} = \left(\begin{array}{cc|cc|cc} -(\lambda_1+\mu_1) & \mu_1 & \lambda_1 & 0 & 0 & 0 \\ 0 & -\lambda_1 & 0 & \lambda_1 & 0 & 0 \\ \hline 0 & 0 & -\lambda_2 & 0 & \lambda_2 & 0 \\ 0 & 0 & 0 & -\lambda_2 & 0 & \lambda_2 \\ \hline \lambda_3 & 0 & 0 & 0 & -(\lambda_3+\mu_2) & \mu_2 \\ \lambda_s(1-\pi) & \lambda_3 & \lambda_s\pi & 0 & 0 & -(\lambda_3+\lambda_s) \end{array}\right)$$

Pour stocker le générateur d'un réseau d'automates stochastiques sous forme tensorielle, nous ne devons garder en mémoire que des petites matrices de dimension n_i, où n_i est le nombre d'états locaux de l'automate $\mathcal{A}^i$. En principe, nous n'avons jamais besoin de calculer explicitement le générateur Q, nous exploitons la forme tensorielle dans les méthodes numériques de résolution.

Ainsi, cette représentation du générateur permet d'avoir une représentation très structurée du modèle. D'autres techniques permettent de stocker le générateur d'un SAN sans utiliser d'algèbre de Kronecker. Nous pouvons par exemple ne garder en mémoire que les éléments non nuls de la matrice et leur position [90]. Nous évoquons par la suite d'autres solutions qui peuvent être adoptées pour représenter le générateur.

III. FORMALISME DES RÉSEAUX D'AUTOMATES STOCHASTIQUES

Maintenant que les principaux concepts de base des réseaux d'automates stochastiques ont été introduits de façon informelle, nous présentons un ensemble de notations et de définitions qui seront ensuite utilisées tout au long de cette thèse.

Nous commençons par donner quelques définitions générales, puis nous nous intéressons plus précisément dans un premier temps aux SANs à temps continu, puis aux SANs à temps discret.

III.1. DÉFINITIONS GÉNÉRALES

Nous considérons la formalisation d'un **réseau d'automates stochastiques** comportant N automates stochastiques $\mathcal{A}^{(i)}$, $i = 1..N$, et un ensemble d'événements $\mathcal{E}$. C'est une

chaîne de Markov à N composantes, et ces composantes ne sont pas forcément indépendantes (présence possible de taux fonctionnels et d'événements synchronisant).

Un **automate stochastique** $\mathcal{A}^{(i)}$ est constitué de :
- un ensemble d'états locaux $S^{(i)}$;
- un ensemble de transitions $T^{(i)}$.

n_i est le nombre des états locaux de l'automate ($n_i = |S^{(i)}|$), et $x^{(i)} \in S^{(i)}$ est un **état local** de l'automate.

L'**espace d'états produit** du SAN est $\hat{S} = S^{(1)} \times ... \times S^{(N)}$. Un **état global** du SAN est un vecteur $x = (x^{(1)}, ..., x^{(N)})$. $x \in \hat{S}$, l'espace d'états produit est l'ensemble des états globaux du SAN. Le nombre d'états globaux est $|\hat{S}| = \prod_{i=1}^{N} n_i$.

Nous passons maintenant à des définitions spécifiques aux SANs à temps continu, puis à temps discret.

III.2. SAN À TEMPS CONTINU

Pour présenter les SAN à temps continu, nous commencerons par donner les définitions de base. Enfin, nous formaliserons la notion de descripteur du SAN.

III.2.1. DÉFINITIONS DE BASE

Les transitions dans les automates sont déclenchées par l'occurrence des événements de $\mathcal{E}$. Nous allons maintenant détailler ces notions d'événement et de transition.

Définition 1 *Un* **événement** $e \in E$ *est défini par :*
- *un taux de franchissement λ_e, qui est une fonction de $\hat{S}$ dans $\mathbb{R}^+$;*
- *un ensemble d'automates impliqués par cet événement O_e (l'occurrence de e entraîne une modification de l'état local de chacun de ces automates) ;*
- *un automate maître $A^{(\eta_e)} \in O_e$.*

L'événement e est **local** *à l'automate $A^{(\eta_e)}$ si O_e est réduit à un automate $A^{(\eta_e)}$. Sinon, l'événement est* **synchronisant**.

Notons qu'un taux de franchissement constant peut être vu comme une fonction qui a toujours la même évaluation, quels que soient les arguments.

Le choix de l'automate maître est arbitraire[5], il sert uniquement à faciliter la compréhension de la structure du descripteur par la suite (section III.2.2).

Définition 2 *Pour chaque automate $\mathcal{A}^{(i)}$, une* **transition** $t^{(i)}$ *est définie par :*
- *un état de départ $x_{t^{(i)},dep} \in S^{(i)}$ et un état d'arrivée $x_{t^{(i)},arr} \in S^{(i)}$;*

[5] Se référer à la section II.3 qui présente le descripteur de façon informelle.

- *un événement associé à la transition* $evt_{t^{(i)}} = e \in \mathcal{E}$ *;*
- *une probabilité de routage* $\pi_{t^{(i)}}$*, qui est une fonction de* $\hat{S}$ *dans* $[0,1]$*. Si* $x \in \hat{S}$*,* $\pi_{t^{(i)}}(x)$ *est la fonction évaluée pour l'état* x*.*

$T^{(i)}$ *est* **l'ensemble** *des transitions.*

Notons qu'un même événement e peut être associé à plusieurs transitions. C'est le cas d'un événement synchronisant, ou d'un événement possible dans plusieurs états locaux distincts d'un même automate (donc générant plusieurs transitions dans cet automate).

De plus, pour tout état global $x = (x^{(1)}, ..., x^{(N)})$ et tout événement $e \in \mathcal{E}$, si nous considérons le sous ensemble des transitions $t^{(i)} \in T^{(i)}$ telles que $x_{t^{(i)},dep} = x^{(i)}$ et $evt_{t^{(i)}} = e$, alors la somme des probabilités de routage correspondant à ces transitions, évaluées pour l'état x, doit être égale à 1.

Il faut également remarquer que, dans certains cas, plusieurs événements peuvent conduire à partir d'un état de départ donné dans un même état d'arrivée. Dans ce cas, l'arc du graphe représentant le SAN est étiqueté par une liste d'événements, mais formellement nous définissons une transition de $T^{(i)}$ pour chaque événement de la liste.

Définition 3 *Dans un état global* $x = (x^{(1)}, ..., x^{(N)}) \in \hat{S}$*,* **l'événement** $e \in \mathcal{E}$ **est possible** *si et seulement si, pour tout automate* $\mathcal{A}^{(i)} \in O_e$*, il existe au moins une transition* $t^{(i)} \in T^{(i)}$ *telle que* $x_{t^{(i)},dep} = x^{(i)}$ *et* $evt_{t^{(i)}} = e$*.*
Dans ce cas, **l'événement** e **a lieu** *avec un taux* $\lambda_e(x)$ *(taux fonctionnel évalué pour l'état de départ), et le nouvel état du SAN* $x' = (x'^{(1)}, ..., x'^{(N)}) \in \hat{S}$ *est défini par :*

- *pour tout* $\mathcal{A}^{(i)} \in O_e$*, pour chaque transition* $t^{(i)}$ *telle que* $x_{t^{(i)},dep} = x^{(i)}$ *et* $evt_{t^{(i)}} = e$*, la probabilité que* $x'^{(i)} = x_{t^{(i)},arr}$ *est* $\pi_{t^{(i)}}(x)$ *;*
- *pour les autres automates* $\mathcal{A}^{(i)}$*,* $x'^{(i)} = x^{(i)}$*.*

L'événement e **est réalisable** *si et seulement si il est possible et son taux de franchissement* $\lambda_e(x)$ *n'est pas nul. Les nouveaux états du SAN possibles lorsque* e *a lieu (correspondant à des probabilités de routage non nulles) sont les* **états successeurs** *de* x *par l'événement* e*.*

Du fait des taux fonctionnels et des événements synchronisant, certains états de $\hat{S}$ ne sont pas accessibles. Si nous reprenons le SAN de la figure 2.2 page 31 (exemple de l'exlusion mutuelle avec des événements locaux), l'état *(activité,activité)* n'est pas accessible.

Définition 4

- *La* **fonction d'accessibilité** F *d'un SAN est une fonction de* $\hat{S}$ *dans* $\{0,1\}$ *;*
- *un état global* x *est* **accessible** *si et seulement si* $F(x) = 1$ *;*
- S *est* **l'ensemble des états accessibles** *du SAN ;* $S \subset \hat{S}$*.*

F possède certaines propriétés, nous ne pouvons pas définir n'importe quelle fonction d'accessibilité pour n'importe quel SAN. Nous choisissons arbitrairement (c'est la modélisation du système qui détermine souvent ce choix) un état **initial** $x \in \hat{S}$, qui est accessible ($x \in S$). A partir de cet état, nous pouvons calculer les états successeurs de x, à savoir tous les états $x' \in \hat{S}$ pour lesquels il existe un événement réalisable e tel que x' est un successeur de x par l'événement e (cf définition 4). Ces états sont dans S.

Nous pouvons ainsi *calculer* F itérativement. Cette fonction d'accessibilité peut également être *donnée* lors de la définition d'un SAN, mais il faut alors s'assurer de son exactitude.

La fonction d'accessibilité F est **bien définie** si et seulement si l'ensemble des états accessibles S forment un graphe de transition fortement connexe.

A partir d'un SAN, nous pouvons construire un unique automate équivalent, dont les états sont uniquement les états accessibles du SAN. Sa matrice de transition est le générateur de la chaîne de Markov correspondant au SAN. La notion de fonction d'accessibilité bien définie assure l'irréductibilité de cette chaîne de Markov sur l'ensemble des états accessibles, et nous pouvons alors employer les théorèmes standards.

Nous détaillons maintenant comment construire le descripteur.

III.2.2. DESCRIPTEUR MARKOVIEN

Le descripteur Markovien est une formule algébrique qui permet une écriture compacte du générateur infinitésimal de la chaîne de Markov correspondante à un SAN par le biais d'une formulation mathématique [80, 2, 41]. Cette formule mathématique décrit, à partir de la description de chaque automate, le générateur infinitésimal de la chaîne de Markov associée au SAN.

Le comportement de chaque automate $\mathcal{A}^{(i)}, i = 1..N$, est décrit par un ensemble de matrices carrées, toutes de dimension n_i. Soit $\mathcal{ES}$ l'ensemble des événements synchronisants ($\mathcal{ES} \subset \mathcal{E}$). Alors, pour $i = 1..N$ et $e \in \mathcal{ES}$, nous donnons quelques notations :

✍ **Soit**

$Q_l^{(i)}$ la matrice contenant les taux de franchissement correspondant aux événements locaux de l'automate $\mathcal{A}^{(i)}$, c'est la matrice de transition locale ;

$Q_{e^+}^{(i)}$ la matrice de synchronisation positive de $\mathcal{A}^{(i)}$, qui représente l'événement synchronisant e, et ses taux de franchissement si $\mathcal{A}^{(i)}$ est le maître de l'événement ;

$Q_{e^-}^{(i)}$ la matrice de synchronisation négative de $\mathcal{A}^{(i)}$, qui correspond à une mise à jour des éléments diagonaux pour l'événement e.

Nous associons donc à chaque automate $1 + 2|\mathcal{ES}|$ matrices.

✍ **Soit**

$Q_j^{(i)}(x^{(i)}, y^{(i)})$ l'élément de la matrice $Q_j^{(i)}$ dans la ligne $x^{(i)}$ et la colonne $y^{(i)}$, avec $i \in [1..N]$ et $j \in \{l, e^+, e^-\}$;

I_{n_i} la matrice identité de taille n_i, avec $i \in [1..N]$;

Définition 5 *Considérons l'automate $\mathcal{A}^{(i)}$ et deux états locaux $x^{(i)}, y^{(i)} \in S^{(i)}$. Le* **taux local de transition** *de $x^{(i)}$ vers $y^{(i)}$ est noté $\tau_l[x^{(i)}, y^{(i)}]$.*

Notons $TL^{(i)}(x^{(i)}, y^{(i)})$ le sous ensemble de l'ensemble des transitions $T^{(i)}$ tel que pour toute transition $t^{(i)} \in TL^{(i)}(x^{(i)}, y^{(i)})$, on ait $x_{t^{(i)},dep} = x^{(i)}$, $x_{t^{(i)},arr} = y^{(i)}$ et $O_{evt_{t^{(i)}}} = \{\mathcal{A}^{(i)}\}$. C'est le sous ensemble des transitions de l'automate $\mathcal{A}^{(i)}$ qui vont de l'état $x^{(i)}$ vers l'état $y^{(i)}$ par un événement local. Alors

$$\tau_l[x^{(i)}, y^{(i)}] = \sum_{t^{(i)} \in TL^{(i)}(x^{(i)}, y^{(i)})} \lambda_{evt_{t^{(i)}}} \times \pi_{t^{(i)}}$$

Notons que lorsqu'il n'existe pas d'arc étiqueté par un événement local allant de $x^{(i)}$ vers $y^{(i)}$ dans le graphe de l'automate $\mathcal{A}^{(i)}$, $\tau_l[x^{(i)}, y^{(i)}] = 0$ (somme sur un ensemble vide).

Définition 6 *Les éléments de la matrice de transition locale de l'automate $\mathcal{A}^{(i)}$ sont définis par :*

6.1. $\forall x^{(i)}, y^{(i)} \in S^{(i)} \mid x^{(i)} \neq y^{(i)}$

$$Q_l^{(i)}(x^{(i)}, y^{(i)}) = \tau_l[x^{(i)}, y^{(i)}] \text{ ;}$$

6.2. $\forall x^{(i)} \in S^{(i)}$

$$Q_l^{(i)}(x^{(i)}, x^{(i)}) = - \sum_{y^{(i)} \in S^{(i)}, y^{(i)} \neq x^{(i)}} \tau_l[x^{(i)}, y^{(i)}] \text{ ;}$$

La définition 6.1 correspond aux éléments non diagonaux de la matrice de transition locale (taux des événements locaux), tandis que la définition 6.2 correspond aux éléments diagonaux (ajustement diagonal des taux des événements locaux).

Nous nous intéressons maintenant aux événements synchronisant.

Définition 7 *Considérons l'automate $\mathcal{A}^{(i)}$, l'événement $e \in \mathcal{ES}$ et deux états locaux $x^{(i)}, y^{(i)} \in S^{(i)}$. Le* **taux synchronisant de transition** *associé à l'événement e de $x^{(i)}$ vers $y^{(i)}$ est noté $\tau_e[x^{(i)}, y^{(i)}]$.*
S'il existe une transition $t^{(i)} \in T^{(i)}$, telle que $x_{t^{(i)},dep} = x^{(i)}$, $x_{t^{(i)},arr} = y^{(i)}$ et $evt_{t^{(i)}} = e$, alors

- *Si $i = \eta_e$ ($\mathcal{A}^{(i)}$ est l'automate maître), $\tau_e[x^{(i)}, y^{(i)}] = \lambda_e \times \pi_t^{(i)}$;*
- *Sinon, $\tau_e[x^{(i)}, y^{(i)}] = \pi_t^{(i)}$.*

Sinon, s'il n'existe pas de transition étiquetée par l'événement synchronisant e allant de $x^{(i)}$ vers $y^{(i)}$ dans l'automate $\mathcal{A}^{(i)}$, $\tau_e[x^{(i)}, y^{(i)}] = 0$.

Définition 8 *Les matrices de synchronisation positive (représentant l'occurrence de l'événement $e \in \mathcal{ES}$) sont définies par :*

8.1. $\forall \mathcal{A}^{(i)} \notin O_e$

$$Q_{e^+}^{(i)} = I_{n_i} \text{ ;}$$

8.2. $\forall \mathcal{A}^{(i)} \in O_e, \forall x^{(i)}, y^{(i)} \in S^{(i)}$

$$Q_{e^+}^{(i)}(x^{(i)}, y^{(i)}) = \tau_e[x^{(i)}, y^{(i)}] \text{ ;}$$

La définition 8.1 correspond aux automates non concernés par l'événement synchronisant e. La définition 8.2 correspond aux automates concernés par l'événement e. La même formulation est utilisée pour tous les automates concernés ($\mathcal{A}^{(i)} \in O_e$, qu'ils soient maître ou esclaves. C'est la définition du taux synchronisant $\tau_e[x^{(i)}, y^{(i)}]$ qui diffère. Nous remarquons qu'il n'y a pas de somme comme pour les événements locaux (définition 5). En effet, nous associons dans la définition 7 un taux par événement, et non pas un taux pour tous les événements comme pour les événements locaux.

Définition 9 *Les matrices de synchronisation négative (représentant l'ajustement arithmétique nécessaire pour construire un générateur), pour $e \in \mathcal{ES}$, sont définies par :*

9.1. $\forall \mathcal{A}^{(i)} \notin O_e$
$$Q_{e^-}^{(i)} = I_{n_i} ;$$

9.2. $\forall x^{(\eta_e)} \in S^{(\eta_e)}$
$$Q_{e^-}^{(\eta_e)}(x^{(\eta_e)}, x^{(\eta_e)}) = \sum_{y^{(\eta_e)} \in S^{(\eta_e)}, y^{(\eta_e)} \neq x^{(\eta_e)}} \tau_e[x^{(\eta_e)}, y^{(\eta_e)}] ;$$

9.3. $\forall \mathcal{A}^{(i)} \in O_e$ *tel que* $i \neq \eta_e$, $\forall x^{(i)} \in S^{(i)}$
$$Q_{e^-}^{(i)}(x^{(i)}, x^{(i)}) = \sum_{y^{(i)} \in S^{(i)}, y^{(i)} \neq x^{(i)}} \tau_e[x^{(i)}, y^{(i)}] ;$$

9.4. $\forall \mathcal{A}^{(i)} \in O_e$, $\forall x^{(i)}, y^{(i)} \in S^{(i)}$ *et* $x^{(i)} \neq y^{(i)}$
$$Q_{e^-}^{(i)}(x^{(i)}, y^{(i)}) = 0 ;$$

La définition 9.1 correspond aux automates non concernés par l'événement synchronisant e. La définition 9.2 correspond aux éléments diagonaux de la matrice de l'automate maître de l'événement e, alors que la définition 9.3 correspond aux automates esclaves. La définition 9.4 correspond aux éléments non diagonaux des matrices des automates concernés par l'événement synchronisant e. Ces éléments sont tous nuls, ce sont des matrices diagonales.

Définition 10 *Le générateur Markovien Q correspondant à la chaîne de Markov associée à un SAN est défini par la formule tensorielle appelée Descripteur Markovien [80, 2, 41] :*

$$Q = \bigoplus_{i=1}^{N}{}_g \, Q_l^{(i)} + \sum_{e \in \mathcal{ES}} \left(\bigotimes_{i=1}^{N}{}_g \, Q_{e^+}^{(i)} + \bigotimes_{i=1}^{N}{}_g \, Q_{e^-}^{(i)} \right) \tag{2.7}$$

Étant donné que toute somme tensorielle est équivalente à une somme de produits tensoriels particuliers (cf section I.2 page 27), le descripteur peut être décrit par :

$$Q = \sum_{j=1}^{(N+2|\mathcal{ES}|)} \bigotimes_{i=1}^{N}{}_g \, Q_j^{(i)} \tag{2.8}$$

$|\mathcal{ES}|$ est le nombre d'événements synchronisant, nous utilisons comme ensemble d'identificateurs des événements synchronisants $\mathcal{ES}$ l'intervalle $[1..E]$, où $E = |\mathcal{ES}|$.

$$\text{Alors } Q_j^{(i)} = \begin{cases} I_{n_i} & \text{pour } j \leq N \text{ et } j \neq i\,; \\ Q_l^{(i)} & \text{pour } j \leq N \text{ et } j = i\,; \\ Q^{(i)}_{(j-N)^+} & \text{pour } N < j \leq N + E\,; \\ Q^{(i)}_{(j-(N+E))^-} & \text{pour } j > N + E\,; \end{cases}$$

Le tableau 2.1 représente les matrices de transition nécessaires à l'écriture de l'équation 2.8. Le logiciel PEPS (section I du chapitre 5) utilise ce tableau comme modèle pour le stockage informatique du descripteur.

$\sum$	N		$Q_l^{(1)} \underset{g}{\otimes} I_{n_2} \underset{g}{\otimes} \cdots \underset{g}{\otimes} I_{n_{N-1}} \underset{g}{\otimes} I_{n_N}$ $I_{n_1} \underset{g}{\otimes} Q_l^{(2)} \underset{g}{\otimes} \cdots \underset{g}{\otimes} I_{n_{N-1}} \underset{g}{\otimes} I_{n_N}$ $\vdots$ $I_{n_1} \underset{g}{\otimes} I_{n_2} \underset{g}{\otimes} \cdots \underset{g}{\otimes} Q_l^{(N-1)} \underset{g}{\otimes} I_{n_N}$ $I_{n_1} \underset{g}{\otimes} I_{n_2} \underset{g}{\otimes} \cdots \underset{g}{\otimes} I_{n_{N-1}} \underset{g}{\otimes} Q_l^{(N)}$
	$2E$	e^+	$Q_{1^+}^{(1)} \underset{g}{\otimes} Q_{1^+}^{(2)} \underset{g}{\otimes} \cdots \underset{g}{\otimes} Q_{1^+}^{(N-1)} \underset{g}{\otimes} Q_{1^+}^{(N)}$ $\vdots$ $Q_{E^+}^{(1)} \underset{g}{\otimes} Q_{E^+}^{(2)} \underset{g}{\otimes} \cdots \underset{g}{\otimes} Q_{E^+}^{(N-1)} \underset{g}{\otimes} Q_{E^+}^{(N)}$
		e^-	$Q_{1^-}^{(1)} \underset{g}{\otimes} Q_{1^-}^{(2)} \underset{g}{\otimes} \cdots \underset{g}{\otimes} Q_{1^-}^{(N-1)} \underset{g}{\otimes} Q_{1^-}^{(N)}$ $\vdots$ $Q_{E^-}^{(1)} \underset{g}{\otimes} Q_{E^-}^{(2)} \underset{g}{\otimes} \cdots \underset{g}{\otimes} Q_{E^-}^{(N-1)} \underset{g}{\otimes} Q_{E^-}^{(N)}$

TAB. 2.1 – Descripteur Markovien

III.3. SAN À TEMPS DISCRET

Nous commençons par une introduction pour motiver l'intérêt des SANs à temps discret, et expliquer les différences fondamentales entre le temps discret et le temps continu. Ensuite, nous donnons les définitions de base des SANs à temps discret, de façon analogue à la section III.2.1. Enfin, nous expliquons comment construire la chaîne de Markov représentant le système, et prouvons que la chaîne obtenue est bien une chaîne de Markov, ce qui justifie le choix de sémantique effectué.

III.3.1. MOTIVATION

La plupart de la recherche sur des approches structurées pour l'évaluation des performances ont supposé une échelle de temps continu. Moins d'attention a été portée sur les systèmes à temps discret. Ces derniers sont en effet plus difficile à modéliser, car plusieurs événements peuvent avoir lieu pendant une même unité de temps (événements concurrents). Ceci n'arrive jamais en temps continu, et les principaux résultats dans ce domaine de recherche ne s'étendent pas pour les modèles à temps discret.

Dans la littérature, nous trouvons cependant un certain nombre de propositions de formalismes pour modéliser des systèmes à temps discret. Différentes sémantiques sont alors proposées, comme nous l'avons détaillé dans l'introduction (chapitre 1 page 11). Nous nous positionnons dans ce chapitre uniquement au niveau des réseaux d'automates stochastiques.

Un formalisme de réseaux d'automates stochastiques à temps discret a été proposé dans [2, 54]. Dans cette approche, la notion d'*événements compatibles* a été introduite pour identifier les événements pouvant se réaliser simultanément pendant une même unité de temps. L'ensemble des événements compatibles doit alors être donné dans le modèle SAN, il est supposé connu. De plus, une contrainte globale sur la somme des probabilités (inférieure à 1) doit être vérifiée pour s'assurer que le modèle est correct. Cependant, la sémantique en cas de conflit ne précise pas quel événement va avoir lieu en premier, et ceci peut engendrer plusieurs comportements différents du modèle. Ainsi, si l'on a deux événements compatibles avec des taux fonctionnels, il se peut que l'un d'eux se réalise et qu'alors le deuxième devienne impossible.

Nous présentons une refonte du formalisme de SAN à temps discret, en recherchant d'une part la simplicité du modèle (le formalisme de [2] est relativement complexe), et d'autre part en détaillant bien la sémantique en cas de conflit et en donnant un algorithme de génération de la chaîne de Markov, en identifiant les événements concurrents automatiquement.

Les définitions que nous avons données pour formaliser les SANs à temps continu doivent être modifiées pour s'adapter au cas des SANs à temps discret. Déjà, il n'y a plus de *taux de franchissement*, mais des *probabilités de transition*. A chaque intervalle de temps, tout événement peut soit avoir lieu, soit ne pas avoir lieu : la loi d'occurrence d'un événement est une variable aléatoire indépendante de Bernouilli.

La différence principale avec les SANs à temps continu est liée à la simultanéité possible des événements dans les modèles en temps discret. Nous devons alors choisir la sémantique à appliquer lorsque cela est possible.

Si nous considérons par exemple le problème de l'exclusion mutuelle exposé en section II.1.1, nous pouvons envisager qu'il y ait dans un même intervalle de temps une relâche de la ressource suivie d'une prise de ressource par un autre client. Nous choisissons pour notre part de donner des priorités aux événements, et d'effectuer les événements dans l'ordre de priorité tant qu'ils sont possibles. Cela n'était pas le cas dans l'approche de [2].

Nous formalisons maintenant les SANs à temps discret.

III.3.2. DÉFINITIONS DE BASE

Dans le cas des SANs à temps discret, les définitions précédentes sont légèrement modifiées, pour prendre en compte la différence de modélisation.

Définition 11 *Un* **événement** $e \in E$ *est défini par :*
- *une probabilité de transition* pt_e*, qui est une fonction de* $\hat{S}$ *dans* $[0, 1]$ *;*
- *une priorité* $prio_e$*, qui détermine comment agir en cas de conflit. C'est un entier strictement positif, 1 est la priorité maximum et* $+\infty$ *la priorité minimum.*
- *un ensemble d'automates impliqués par cet événement* O_e *(l'occurrence de* e *entraîne une modification de l'état local de chacun de ces automates).*

L'événement e *est* **local** *à l'automate* A *si* O_e *est réduit à un automate* A*. Sinon, l'événement est* **synchronisant**.

Notons qu'une probabilité de transition constante peut être vue comme une fonction qui a toujours la même évaluation, quels que soient les arguments. La loi d'occurrence d'un événement est une variable aléatoire indépendante de Bernouilli.

Pour permettre la simultanéité des événements, nous voulons pouvoir rendre un événement possible dans certains états même si cet événement ne peut en fait avoir lieu que si un autre événement plus prioritaire a eu lieu avant. Nous introduisons pour cela un état particulier qui peut être adjoint à tout automate :

Définition 12 $\emptyset$ *est appelé l'***état vide**.

Définition 13 *Pour chaque automate* $\mathcal{A}^{(i)}$*, une* **transition** $t^{(i)}$ *est définie par :*
- *un état de départ* $x_{t^{(i)},dep} \in S^{(i)}$ *et un état d'arrivée* $x_{t^{(i)},arr} \in S^{(i)} \cup \emptyset$ *;*
- *un événement associé à la transition* $evt_{t^{(i)}} = e \in \mathcal{E}$ *;*
- *une probabilité de routage* $\pi_{t^{(i)}}$*, qui est une fonction de* $\hat{S}$ *dans* $[0, 1]$*. Si* $x \in \hat{S}$*,* $\pi_{t^{(i)}}(x)$ *est la fonction évaluée pour l'état* x*.*

$T^{(i)}$ *est* **l'ensemble** *des transitions.*

Notons qu'un même événement e peut être associé à plusieurs transitions. C'est le cas d'un événement synchronisant, ou d'un événement possible dans plusieurs états locaux distincts d'un même automate (donc générant plusieurs transitions dans cet automate).

De plus, pour tout état global $x = (x^{(1)}, ..., x^{(N)})$ et tout événement $e \in \mathcal{E}$, si nous considérons le sous ensemble des transitions $t^{(i)} \in T^{(i)}$ telles que $x_{t^{(i)},dep} = x^{(i)}$, $x_{t^{(i)},arr} \in S^{(i)}$ et $evt_{t^{(i)}} = e$, alors la somme des probabilités de routage correspondant à ces transitions, évaluées pour l'état x, doit être égale à 1. Ces probabilités de routage sont des variables aléatoires indépendantes de toutes les autres variables aléatoires du modèle.

Une transition ne peut mener à l'état vide que s'il n'existe pas d'autres transitions définies par le même état de départ et le même événement, dont l'état d'arrivée est différent de l'état vide. En effet, dans ce dernier cas, l'événement est possible depuis l'état de départ donc il est inutile de rajouter une transition menant à l'état vide, cette dernière ne servant qu'à rendre possible l'événement depuis l'état de départ.

Il faut également remarquer que, dans certains cas, plusieurs événements peuvent conduire à partir d'un état de départ donné dans un même état d'arrivée. Dans ce cas, l'arc du graphe représentant le SAN est étiqueté par une liste d'événements, mais formellement nous définissons une transition de $T^{(i)}$ pour chaque événement de la liste.

Définition 14 *Dans un état global $x = (x^{(1)}, ..., x^{(N)}) \in \hat{S}$,* **l'événement** $e \in \mathcal{E}$ **est possible** *si et seulement si, pour tout automate $\mathcal{A}^{(i)} \in O_e$, il existe au moins une transition $t^{(i)} \in T^{(i)}$ telle que $x_{t^{(i)},dep} = x^{(i)}$ et $evt_{t^{(i)}} = e$.*
Dans ce cas, e **est réalisable** *depuis x si et seulement si, pour tout automate $\mathcal{A}^{(i)} \in O_e$, il existe au moins une transition $t^{(i)} \in T^{(i)}$ telle que $x_{t^{(i)},dep} = x^{(i)}$, $evt_{t^{(i)}} = e$, $x_{t^{(i)},arr} \in S^{(i)}$, et $pt_e(x) \neq 0$.*
Soit $EP(x)$ **l'ensemble des événements possibles**[6] *dans l'état x.*
L'événement $e \in EP(x)$ réalisable qui a la priorité maximum dans cet ensemble **a lieu** *avec la probabilité $pt_e(x)$ (probabilité de transition fonctionnelle évaluée pour l'état de départ), et le nouvel état du SAN $x' = (x'^{(1)}, ..., x'^{(N)}) \in \hat{S}$ est défini par :*

- *pour $\mathcal{A}^{(i)} \in O_e$, pour chaque transition $t^{(i)}$ telle que $x_{t^{(i)},dep} = x^{(i)}$ et $evt_{t^{(i)}} = e$, nous avons $x'^{(i)} = x_{t^{(i)},arr}$ avec la probabilité de routage $\pi_{t^{(i)}}(x)$;*
- *pour les autres automates $\mathcal{A}^{(i)}$, nous avons $x'^{(i)} = x^{(i)}$.*

x' est un **état intermédiaire**.

Dans le même intervalle de temps, tous les événements de $EP(x)$ sont susceptibles d'avoir lieu, du plus prioritaire au moins prioritaire. Après avoir atteint l'état intermédiaire x', nous considérons que $EP(x)$ est diminué de l'événement e qui a eu lieu. Nous recommençons alors cette procédure tant qu'il y a des événements dans $EP(x)$, pour l'événement le plus prioritaire, à partir de l'état intermédiaire x'. A chaque étape, l'état de départ que nous examinons est appelé l'*état courant*.

Une transition vers l'état vide permet de rendre un événement possible dans un état donné, sans qu'il soit réalisable. Ainsi, il pourra avoir lieu uniquement si un événement plus prioritaire a lieu avant, et si l'événement est réalisable depuis l'état intermédiaire atteint par l'occurrence de ce dernier événement. Considérons par exemple une file d'attente dans laquelle les départs sont plus prioritaires que les arrivées. Lorsque la file est pleine, nous

[6] Dans [2], $EP(x)$ correspond à l'ensemble des événements compatibles à partir de x.

pouvons autoriser une arrivée (transition vers l'état vide), mais une arrivée ne sera possible que si un départ a lieu avant. Sinon, la file étant pleine, il ne peut pas y avoir d'arrivée.

Comme pour les modèles à temps continu, certains états peuvent ne pas être accessibles.

Nous détaillons ici comment construire la chaîne de Markov à temps discret qui représente le système. Cette chaîne ne contient que les états accessibles du modèle.

III.3.3. CONSTRUCTION DE LA CHAÎNE DE MARKOV ÉQUIVALENTE

Les états de la chaîne de Markov sont les états accessibles du SAN. C'est un sous-ensemble de $\hat{S}$. Pour construire les transitions de la chaîne de Markov, nous étudions les états de départ les uns après les autres, et nous calculons les probabilités d'aller dans un autre état.

Nous présentons tout d'abord le squelette de l'algorithme qui permet de générer tous les états accessibles à partir d'un état initial donné x, et les probabilités de transition associées. Nous obtenons ainsi toutes les transitions partant de l'état x. Cet état est donc par définition accessible. Au démarrage de l'algorithme, l'état courant $y = x$, et nous examinons alors tous les événements $e \in EP(x)$, du plus au moins prioritaire.

- Si e est réalisable depuis y, il peut avoir lieu ou non, avec les probabilités respectives $pt_e(y)$ et $1 - pt_e(y)$. S'il n'a pas lieu, l'état courant ne change pas. Sinon, nous obtenons une liste d'états intermédiaires qui peuvent être obtenus par différents routages.
- Si e n'est pas réalisable, la probabilité qu'il n'ait pas lieu est 1. On reste donc dans le même état.

A partir de tous les états intermédiaires obtenus (nouveaux états courants), nous traitons les événements de $EP(x)$ qui suivent par ordre de priorité.

Remarquons que nous décidons ainsi de la sémantique lorsque plusieurs événements arrivent dans le même intervalle de temps. Après avoir déclenché le plus prioritaire, on atteint un état intermédiaire, et nous regardons si l'événement de $EP(x)$ suivant (par ordre de priorité) est réalisable depuis cet état.

Lorsque la liste d'événements est épuisée, l'état atteint est un état de la chaîne de Markov, obtenu éventuellement par l'occurrence simultanée de plusieurs événements (les événements de $EP(x)$ qui ont eu lieu). Le problème est de générer tous les états d'arrivées possibles, avec la probabilité qu'on arrive dans chaque état. Nous obtenons ainsi toutes les transitions de la chaîne de Markov partant de l'état x.

Pour définir formellement cela, nous introduisons une nouvelle structure, qui est une liste de couples constitués d'un état et d'une probabilité. Pour une liste donnée, nous disposons donc d'un ensemble d'états accessibles et de la probabilité d'être dans chaque état.

Définition 15 *Une* **liste d'états accessibles pondérés par des probabilités** *est une liste de couples (état, probabilité), notée* $[(s_1, p_1), (s_2, p_2), ...]$.

- $[\,]$ *est la liste vide ;*
- $[(s, p)]$ *est la liste contenant l'état* s *auquel nous associons la probabilité* p *;*

– $[(s_i, p_i)]_{i\in[1..k]}$ *est la liste contenant* k *états* $s_1, ..., s_k$ *auxquels nous associons les probabilités* p_i *(*$i \in [1..k]$*).*

Nous définissons alors des opérations sur ces listes :

Définition 16 *Nous disposons de plusieurs opérations sur les listes d'états accessibles pondérés par des probabilités :*

- *L'***ajout** *d'un couple (état, probabilité) dans la liste :*
  - $(s, p) \cup [\,] = [(s, p)]$
  - $(s_{k+1}, p_{k+1}) \cup [(s_i, p_i)]_{i\in[1..k]} = [(s_i, p_i)]_{i\in[1..k+1]}$
- *La fonction* **combine**$((s, p), L)$ *regarde s'il y a déjà une occurrence de* s *dans* L*, et si c'est le cas, additionne la probabilité de* s *dans* L *avec* p *:*
  - $combine((s, p), [\,]) = [(s, p)]$
  - $combine((s, p), (s', p') \cup L) =$
    si $s = s'$ *alors* $combine((s, p + p'), L)$ *sinon* $(s', p') \cup combine((s, p), L)$
- *La fonction* **compresse** *prend une liste comme paramètre, et ne laisse qu'une seule entrée par état distinct dans la liste résultat.*
  - $compresse([\,]) = [\,]$
  - $compresse((s, p) \cup L) = combine((s, p), compresse(L))$

L'algorithme 2.1 calcule la liste des états accessibles pondérés par leur probabilité à partir d'un état initial x donné. Nous obtenons ainsi éventuellement de nouveaux états accessibles, ce sont des états de la chaîne de Markov. Nous obtenons également toutes les transitions partant de l'état x. Pour obtenir tous les états accessibles du SAN, nous pouvons alors refaire tourner l'algorithme 2.1 en partant des nouveaux états accessibles obtenus. Nous effectuons donc une exploration de l'espace des états accessibles S, et nous obtenons au final toutes les probabilités de transition d'un état à un autre.

Petit exemple pour clarifier l'algorithme 2.1

Nous illustrons maintenant le déroulement de l'algorithme 2.1 sur un petit exemple d'étude pour en clarifier le sens.

Le système considéré est celui de l'exclusion mutuelle déjà évoqué en temps continu (section II.1.1 page 29).

Les événements du modèle en temps discret sont identiques à ceux du modèle en temps continu, et nous pouvons représenter le SAN par la même figure 2.2 (page 31). Seules les caractéristiques de chaque événement diffèrent.

Déjà, nous ne parlons plus de taux de franchissement mais de probabilité de transition. La fonction f garantissant l'unicité de la ressource reste la même que dans l'exemple en temps continu, à savoir elle vaut 1 si et seulement si les deux clients sont dans l'état repos. Les probabilités de prise et de relâche de ressource sont respectivement λ et μ.

Nous devons également préciser les priorités de chaque événement. Dans notre cas, nous considérons que la relâche de ressource est plus prioritaire que la prise de ressource. Ainsi, il se peut que dans une même unité de temps, un client relâche la ressource, et l'autre

Algorithme 2.1

Entrée : un réseau d'automates stochastiques, et un état initial $x = (x^{(1)}, ..., x^{(N)})$.
Sortie : une liste d'états accessibles pondérés par leur probabilité.

1. Initialiser la liste des états à traiter $E = [(x, 1)]$ (au démarrage de l'algorithme, a probabilité d'être dans l'état x est 1).
   Soit L l'ensemble des événements possibles depuis x, $L = EP(x)$.
2. Tant que L n'est pas vide,
   - Choisir un événement $e \in L$ parmi ceux de plus haute priorité, et supprimer e de la liste L.
   - Initialiser la nouvelle liste d'états accessibles $E' = [\,]$ (c'est la liste des états que l'on peut atteindre suivant l'occurrence ou non de e depuis x).
   - Pour tout $(y, p) \in E$,
     - Si e n'est pas réalisable depuis y, $E' = (y, p) \cup E'$ (il n'y a pas de nouveaux états accessibles, on reste dans l'état y).
     - Sinon, si e est réalisable,
       - e n'a pas lieu avec la probabilité $1 - pt_e(y)$, d'où $E' = (y, 1 - pt_e(y)) \cup E'$.
       - Il faut envisager tous les cas où e a lieu, on obtient un ensemble d'états intermédiaires y' ainsi que la probabilité que l'on soit dans cet état ($p' = pt_e(y) \times$ *un produit de probabilités de routage*), et rajouter tous les couples (y', p') à E'. Le produit de probabilités de routage est obtenu en regardant chaque état local possible dans l'état intermédiaire y' pour les automates impliqués par l'événement e, d'après la définition 14.
   - Lorsqu'on a terminé le traitement d'un événement, on "compresse" la liste des états accessibles E' obtenue (cf définition 16) pour éviter d'examiner par la suite plusieurs fois le même état de départ lors de l'étape suivante. La nouvelle liste des états à traiter est donc $E = compresse(E')$.
3. Lorsque L est vide, E est la liste des états accessibles depuis l'état initial x, pondérés par leur probabilité.

Algorithme 2.1: Génération des états accessibles et de leur probabilité

client la prenne. Ceci ne pouvait pas avoir lieu en temps continu. Nous donnons également des priorités différentes aux deux événements de prise de ressource pour pouvoir décider quel client prend la ressource lorsqu'ils la demandent tous les deux en même temps. Pour notre exemple, le premier client est plus prioritaire.

Nous avons alors les événements suivants :

- la prise de la ressource par le premier client, p_1,
  avec $pt_{p_1} = \lambda f$, $prio_{p_1} = 2$, $O_{p_1} = \{\mathcal{A}^{(1)}\}$;
- la prise de la ressource par le deuxième client, p_2,
  avec $pt_{p_2} = \lambda f$, $prio_{p_2} = 3$, $O_{p_2} = \{\mathcal{A}^{(2)}\}$;
- la relâche de la ressource par le premier client, r_1,
  avec $pt_{r_1} = \mu$, $prio_{r_1} = 1$, $O_{r_1} = \{\mathcal{A}^{(1)}\}$;
- la relâche de la ressource par le deuxième client, r_2,
  avec $pt_{r_2} = \mu$, $prio_{r_2} = 1$, $O_{r_2} = \{\mathcal{A}^{(2)}\}$.

Pour construire la chaîne de Markov équivalente à ce modèle, nous appliquons l'algorithme en partant de l'état initial $x = (repos, repos)$.

Le déroulement de l'algorithme peut être représenté de façon arborescente. La racine de l'arbre correspond à l'initialisation de l'algorithme 2.1 (étape 1), puis nous descendons d'un niveau dans l'arbre lors du traitement d'un événement.

A un niveau donné, nous disposons donc de l'ensemble des états accessibles avec leurs priorités, ce sont des états intermédiaires. Nous regardons pour chaque état si l'événement est réalisable ou non depuis cet état, et nous générons les nouveaux états accessibles, tenant compte de la probabilité de transition de l'événement et des probabilités de routage éventuelles. A chaque niveau, les listes L et E sont mises à jour (boucle de l'algorithme 2.1 dans l'étape 2).

Les feuilles de l'arbre correspondent aux éléments de la liste des états accessibles à partir de l'état de départ, pondérés par des probabilités (étape 3 de l'algorithme 2.1).

Pour notre exemple, nous codons l'état *repos* par 1 et l'état *activité* par 2, un état global est alors noté Sij, où $i = 1, 2$ est l'état du premier client, et $j = 1, 2$ est l'état du deuxième client. L'état initial nous donne un certain nombre d'états accessibles, nous relançons alors l'algorithme 2.1 à partir de ces états pour obtenir tous les états accessibles et les probabilités de transition d'un état à un autre. Nous obtenons alors les arbres de la figure 2.6 (un arbre par appel à l'algorithme).

III.3.4. DÉFINITION DE LA CHAÎNE DE MARKOV

Les états de la chaîne obtenue sont donc les états accessibles, et les probabilités de transition ont été données par l'algorithme 2.1. Les probabilités de transition ne dépendent que de l'état de départ, c'est un processus sans mémoire. Pour s'assurer que nous obtenons bien une chaîne de Markov, il reste à vérifier que pour chaque état de la chaîne, la somme des probabilités partant de cet état est égale à 1 (c'est la somme des probabilités sur une ligne de la matrice de transition).

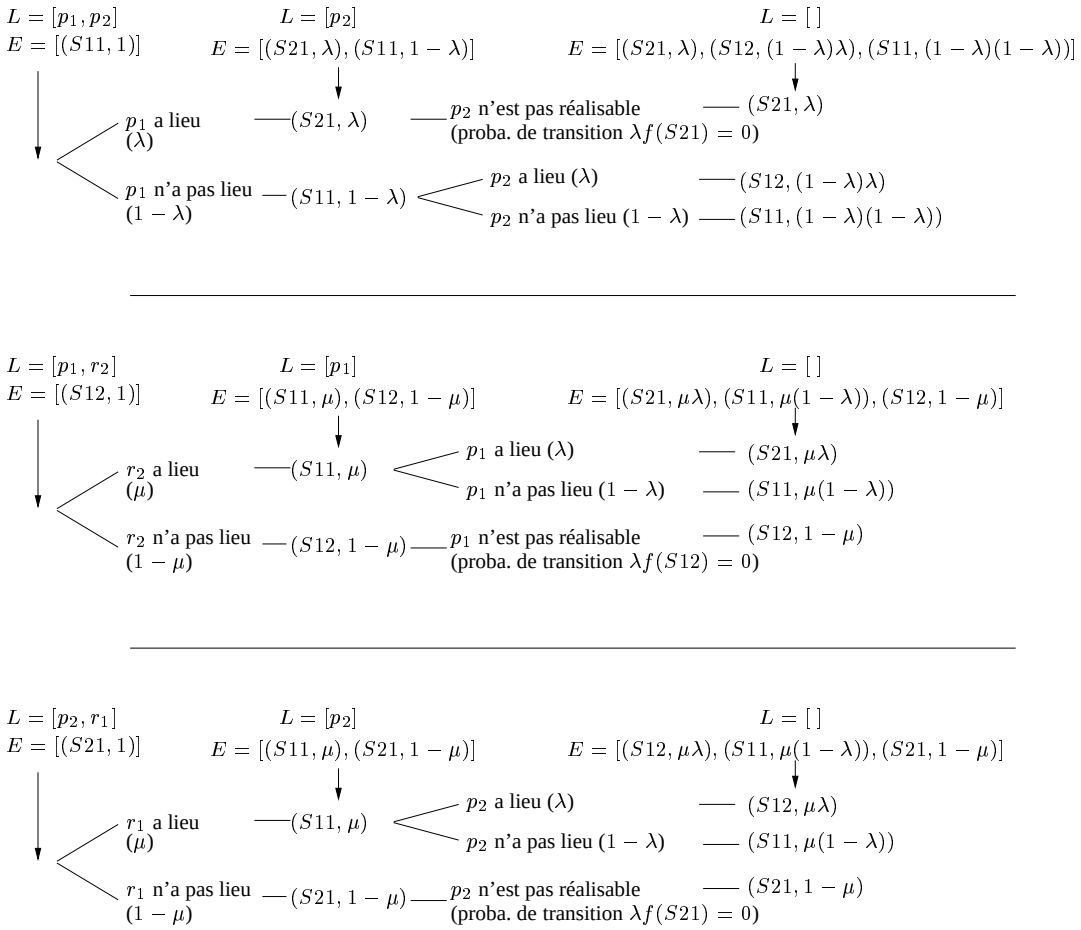


FIG. 2.6 – Déroulement de l'algorithme 2.1

En partant de l'algorithme 2.1, nous montrons qu'à chaque itération, la somme des probabilités des états dans E est égale à 1. Notons $\sum E$ cette somme. La preuve est faite par récurrence.

Au départ, $E = [(x, 1)]$, donc $\sum E = 1$.

Supposons qu'à une étape donnée (une étape correspond au traitement d'un événement de L), nous avons $\sum E = 1$. Nous générons à partir de tout (y, p) de E une liste

1. $E1 = [(y, p)]$ si l'événement n'est pas réalisable, ou bien
2. $E2 = [(y, p(1 - pt_e(y))), (y'_1, p * pt_e(y) * \pi_1(y)), ..., (z_r, p * pt_e(y) * \pi_r(y))]$ si l'événement est réalisable.

Dans le deuxième cas, $\pi_i(y)$ ($i \in [1..r]$) représente la probabilité de routage pour aller dans l'état y'_i (dans le cas d'événements synchronisant, c'est le produit des probabilités de routage dans chaque automate).

Nous avons alors $\sum E1 = p$ et
$\sum E2 = p(1 - pt_e(y)) + p * pt_e(y) * \pi_1(y) + ... + p * pt_e(y) * \pi_z(y)$
$= p(1 - pt_e(y)) + p * pt_e(y) * (\pi_1(y) + ... + \pi_r(y))$

Par définition du routage, $\pi_1(y) + ... + \pi_r(y) = 1$ donc
$\sum E2 = p(1 - pt_e(y)) + p * pt_e(y) = p$.

Dans les deux cas, la somme des probabilités de la liste générée par l'élément (y, p) de E est égale à p. La nouvelle liste E' est l'union de ces listes générées, donc
$\sum E' = \sum E = 1$.

La chaîne obtenue est bien une chaîne de Markov, ce qui prouve la cohérence de la sémantique utilisée.

Reprenons l'exemple de l'exclusion mutuelle présenté dans la section III.3.3. A partir des différents appels à l'algorithme, détaillés dans la figure 2.6, nous pouvons en déduire la chaîne de Markov équivalente (figure 2.7) et sa matrice de transition (états dans l'ordre $S11$, $S12$, $S21$) :

$$Q = \begin{pmatrix} (1-\lambda)(1-\lambda) & \lambda(1-\lambda) & \lambda \\ \mu(1-\lambda) & (1-\mu) & \lambda\mu \\ \mu(1-\lambda) & \lambda\mu & (1-\mu) \end{pmatrix}$$

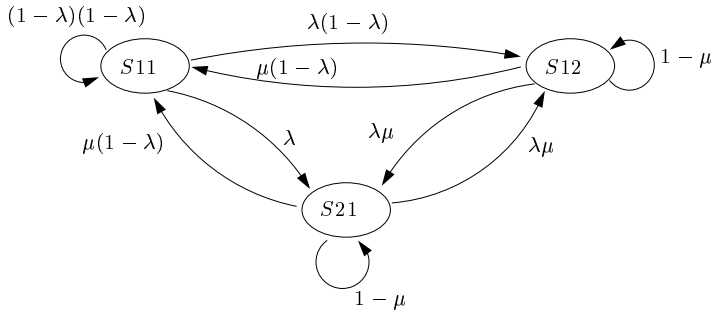


FIG. 2.7 – Chaîne de Markov équivalente – exemple de l'exclusion mutuelle

III.3.5. CONCLUSION

Nous avons ainsi présenté un formalisme de SANs à temps discret, et donné un algorithme de génération de la chaîne de Markov à partir du SAN. Cet algorithme procède en générant l'ensemble des états accessibles à partir d'un état de départ donné, ainsi que les probabilités associées à chaque état accessible. C'est un algorithme similaire à des algorithmes de génération de l'espace d'états accessibles [67, 22, 32], sauf qu'il fonctionne en temps discret et génère au passage les événements concurrents.

Des priorités doivent alors être associées à chaque événement pour déterminer comment agir en cas de conflit. Un problème ouvert reste l'attribution des priorités, qui doit être

effectuée par l'utilisateur pour donner un sens au modèle. Une attribution automatique peut être envisagée pour les gros modèles, consistant à fixer un ordre de priorité sur les automates et sur certains événements, puis à générer automatiquement la priorité de chaque événement à partir de là.

Nous présentons maintenant quelques exemples de modélisation à l'aide de SANs.

IV. EXEMPLES DE MODÉLISATION

Cette section présente différents exemples de modélisation à l'aide de réseaux d'automates stochastiques. Nous illustrons à travers ces exemples les différents types d'interaction entre automates, en modélisant certains systèmes de plusieurs façons.

Nous présentons tout d'abord différentes variantes d'un système de partage de ressources en *temps continu*. Cet exemple classique permet d'aborder la plupart des concepts des réseaux d'automates stochastiques [41, 42, 7].

Des réseaux de files d'attente ouvert avec blocage, perte et priorité sont décrits dans la section IV.2 [7, 6]. Il s'agit ici aussi d'exemples de SAN à *temps continu.*

Enfin, nous exposons la modélisation d'une file d'attente à *temps discret* avec différentes politiques de service [50] dans la section IV.3.

IV.1. PARTAGE DE RESSOURCES : RS

Ce premier exemple est une généralisation du problème de l'exclusion mutuelle, exposé en section II.1.1.

N_p processus, ou clients, se partagent N_r unités identiques d'une ressource commune. Chaque processus peut être dans deux états, un état de *repos*, ou un état *actif* correspondant à l'utilisation d'une unité de ressource. Lorsqu'un processus désirant passer de l'état repos à l'état actif trouve N_r autres processus utilisant la ressource, ce processus ne peut pas accéder à la ressource et il reste dans l'état repos.

Nous pouvons remarquer que lorsque $N_r = 1$, ce problème est celui de l'exclusion mutuelle, et lorsque $N_r = N_p$, tous les processus sont indépendants.

Pour $i = 1..N_p$, soit $\lambda^{(i)}$ le taux avec lequel le processus i désire accéder à une ressource, et soit $\mu^{(i)}$ le taux avec lequel ce même processus relâche la ressource après utilisation.

IV.1.1. MODÉLISATION AVEC FONCTION : RS1

Chaque processus i ($i = 1..N_p$) est modélisé par un automate à deux états $\mathcal{A}^{(i)}$, les deux états étant *repos* et *actif*. Nous définissons alors une fonction f de restriction à l'accès aux ressources[7] :

$$f(x) = \delta\left(\sum_{i=1}^{N_p} \delta(x^{(i)} = \textit{actif}) < N_r\right)$$

où $x^{(i)}$ est l'état local de l'automate $\mathcal{A}^{(i)}$, et $x = (x^{(1)}, \ldots, x^{(N_p)})$ est l'état global du SAN. Ainsi, f vaut 1 lorsque l'accès à la ressource est possible, et 0 lorsque toutes les unités de ressource sont déjà utilisées.

Nous définissons alors deux événements par processus :

– $P^{(i)}$ correspond à la "prise de ressource" par le processus i et il a un taux $\lambda^{(i)} f$;

– $R^{(i)}$ correspond à la "relâche de ressource" par le processus i et il a un taux $\mu^{(i)}$.

Ce modèle, appelé **RS1** (*Resource Sharing*), est illustré dans la figure 2.8.

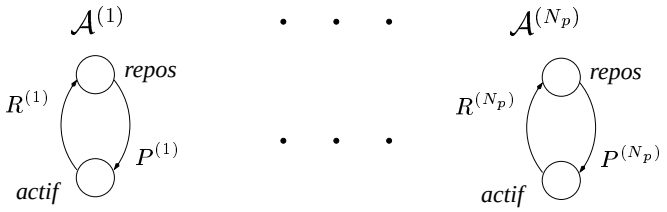


FIG. 2.8 – Partage de ressources, modèle avec fonction – RS1

La fonction d'accessibilité F du SAN est définie de façon analogue à la fonction de restriction à l'accès aux ressources f. Un état est accessible si et seulement si le nombre total de processus en train d'utiliser une unité de ressource est inférieur ou égal au nombre d'unités de ressource disponibles :

$$F(x) = \delta\left(\sum_{i=1}^{N_p} \delta(x^{(i)} = \textit{actif}) \leq N_r\right)$$

Ce modèle ne comprend pas d'événements synchronisants, il y a donc seulement une matrice de transition locale associée à chaque processus :

$$Q_l^{(i)} = \begin{pmatrix} -\lambda^{(i)} f & \lambda^{(i)} f \\ \mu^{(i)} & -\mu^{(i)} \end{pmatrix}$$

Le descripteur Q du SAN est alors défini par la somme tensorielle :

$$Q = \bigoplus_{i=1}^{N}{}_g \, Q_l^{(i)}$$

[7] $\delta(b)$ est une fonction qui vaut 1 si l'expression b est vraie, sinon la fonction vaut 0.

La taille de l'espace d'états produit de ce modèle est égale à 2^{N_p}, et le nombre d'états accessibles est égal à[8] $\sum_{i=0}^{N_r} \binom{N_p}{i}$. De ce fait, nous pouvons noter que pour des valeurs de N_r petites par rapport à N_p (peu de ressources et beaucoup de processus), la proportion d'états accessibles par rapport au nombre total d'états de l'espace produit peut être très petite.

IV.1.2. MODÉLISATION AVEC SYNCHRONISATION : RS2

Nous pouvons modéliser le même système sans utiliser de fonctions, mais uniquement des événements synchronisant.

Dans ce cas, chaque processus i ($i = 1..N_p$) est modélisé par un automate à deux états $\mathcal{A}^{(i)}$, les deux états étant *repos* et *actif*. Un automate supplémentaire $\mathcal{A}^{(N_p+1)}$ modélise l'état de la ressource. Cet automate possède $N_r + 1$ états, et l'état i ($i \in [0..N_r]$) signifie que i unités de ressource sont actuellement utilisées. Les événements de prise et de relâche de ressource sont alors des événements synchronisant le processus avec l'automate ressource. Une prise de ressource doit en effet incrémenter l'état de l'automate ressource, et elle n'est possible que si ce dernier état est différent de N_r. Au contraire, une relâche de ressource implique un décrément de l'état de l'automate ressource.

Ce modèle, appelé **RS2** (*Resource Sharing*), est illustré dans la figure 2.9.

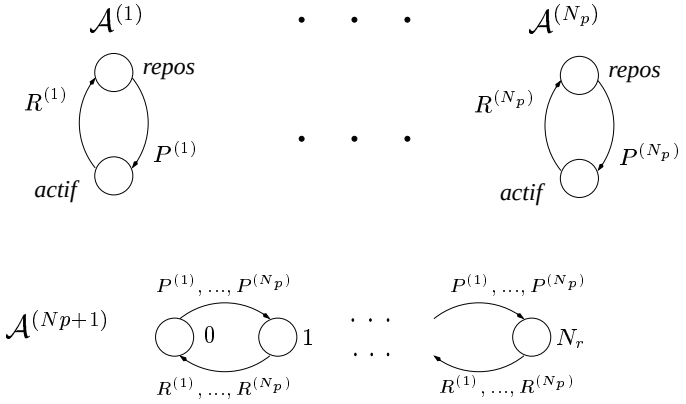


FIG. 2.9 – Partage de ressources, modèle sans fonctions – RS2

Pour ce modèle, un état est accessible si et seulement si le nombre de processus en train d'utiliser une unité de ressource est égal à l'état de l'automate ressource :

$$F(x) = \delta\left(\sum_{i=1}^{N_p} \delta(x^{(i)} = \textit{actif}) = x^{(N_p+1)}\right)$$

$x = (x^{(1)},\ ...,\ x^{(N_p+1)})$ est un état global du SAN.

[8] Notons $\binom{a}{b}$ le nombre de combinaisons non ordonnées de taille b avec a éléments distincts. Sa définition numérique est $\binom{a}{b} = \frac{a!}{(a-b)!b!}$.

Nous associons dans ce cas deux matrices par automate et par événement, et le descripteur est une somme de produits tensoriels.

La taille de l'espace d'états produit de ce modèle est égale à $2^{N_p} \times (N_r + 1)$, et le nombre d'états accessibles est identique à celui du modèle RS1, à savoir $\sum_{i=0}^{N_r} \binom{N_p}{i}$. L'espace d'états produit est plus grand que celui de RS1 pour un même espace d'états accessibles, donc la proportion d'états accessibles sera toujours plus faible que pour le modèle RS1. C'est généralement le cas lorsque nous utilisons des synchronisations à la place des fonctions, nous sommes obligés de rajouter des automates supplémentaires, et les matrices du descripteur sont alors très creuses.

IV.1.3. MODÈLE AVEC PANNE : RS3

Nous présentons ici une variante de RS1. Le système est identique sauf qu'il peut tomber en panne. Chacun des N_p processus a donc un état supplémentaire, l'état *panne*.

L'événement synchronisant *Off* (taux λ_{off}) correspond à une panne du système. L'occurrence de l'événement *On* (taux λ_{on}) signifie que le système a été réparé, et alors tous les processus se retrouvent dans l'état *repos*. Il n'y a pas de mémoire de l'état des processus avant la panne.

Nous modélisons l'état du système par un automate supplémentaire $\mathcal{A}^{(N_p+1)}$, qui a deux états *marche* et *arrêt*. Les transitions d'un état à l'autre sont synchronisées par les événements *Off* et *On*.

Ce modèle, appelé **RS3** (*Resource Sharing*), est illustré dans la figure 2.10.

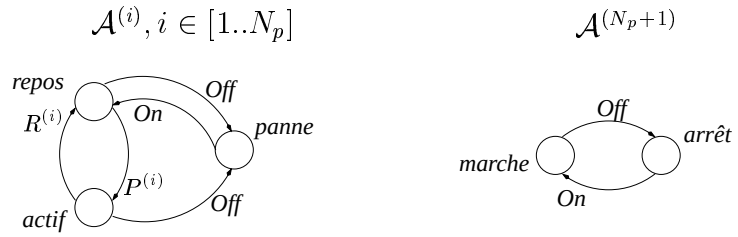


FIG. 2.10 – Partage de ressources, modèle avec panne – RS3

Ce modèle comprend à la fois des événements synchronisant et des événements locaux avec taux de franchissement fonctionnels. Le descripteur comprend donc une partie locale (somme tensorielle généralisée) et une partie synchronisante (produits tensoriels).

La taille de l'espace d'états produit de ce modèle est égale à $2^{N_p} \times 2$. Au niveau du nombre d'états accessibles, il y en a un de plus que pour le modèle RS1, à savoir l'état dans lequel tous les processus sont dans l'état *panne*, et l'automate $\mathcal{A}^{(N_p+1)}$ dans l'état *arrêt*. Nous avons donc $1 + \sum_{i=0}^{N_r} \binom{N_p}{i}$ états accessibles.

IV.2. RÉSEAU DE FILES D'ATTENTE : QN

Le système que nous cherchons à modéliser est un réseau de files d'attente ouvert avec des files d'attente de capacité finie (avec blocage ou perte), différentes classes de clients, des priorités, et des taux de service dépendant du nombre de clients dans d'autres files. Nous considérons deux systèmes différents.

Le premier est un réseau de files d'attente ouvert avec blocage et priorité, **QN1**. Le deuxième, **QN2**, est constitué de quatre files et deux classes de clients.

IV.2.1. QN1

Ce premier réseau de files d'attente est composé de N files de capacité finie, et de $N-1$ classes de clients différents.

Pour $i = 1..N-1$, les clients de classe i arrivent de l'extérieur vers la file i suivant un taux exponentiel λ_i. Les clients sont perdus si la file est pleine. Le serveur de la file i sert les clients suivant un taux exponentiel μ_i.

Les clients qui ont été servis par les files $1..N-1$ sont redirigés vers la file N. Si cette dernière est pleine, les clients sont bloqués, et les serveurs des autres files doivent s'arrêter. Ainsi, ces serveurs ne peuvent pas servir un nouveau client tant que la file N est pleine. Lorsqu'une place se libère dans cette file, les clients bloqués à la sortie des autres files sont transférés vers la file N.

La file N sert les clients de classe i ($i = 1..N-1$) suivant un taux exponentiel μ_N^i. C'est la seule file qui sert les $N-1$ classes de clients. Dans cette file, pour $1 \leq i < j \leq N-1$, les clients de classe i sont prioritaires sur ceux de classe j. Ainsi, un client ne peut être servi par la file N que s'il n'y a aucun client d'une classe prioritaire à lui dans cette file. Après avoir été servis par la file N, les clients quittent le réseau.

Nous notons C_i la capacité de la file i (pour $i = 1..N$). La figure 2.11 illustre ce modèle.

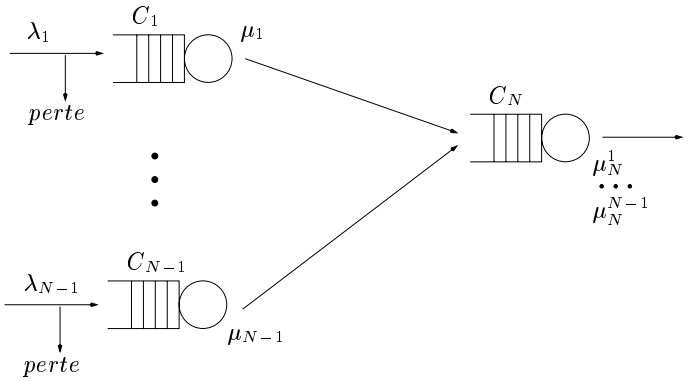


FIG. 2.11 – Réseau de files d'attente – QN1

Les files i, pour $i = 1..N-1$, sont chacune représentées par un unique automate $\mathcal{A}^{(i)}$, avec $C_i + 1$ états. Lorsqu'il y a k clients dans une des files, elle est dans l'état k. La file N nécessite $N-1$ automates pour sa représentation : $\mathcal{A}^{(N,i)}$ correspond au nombre de clients de classe i présents dans la file, pour $i = 1..N-1$.

Les arrivées de clients sont modélisées par des événements locaux a_i ($i = 1..N-1$). L'événement a_i représente l'arrivée d'un client de classe i dans la file i, il a un taux λ_i.

Pour modéliser le transfert des clients d'une file à une autre, on utilise $N-1$ événements synchronisant. Ainsi, s_i ($i = 1..N-1$) correspond à un transfert d'un client de classe i de la file i vers la file N. On définit une fonction f pour restreindre les arrivées dans la file N. Elle a pour valeur 0 quand la file N est pleine, et 1 dans les autres cas :

$$f(\tilde{x}) = \delta\left(\left[\sum_{k=1}^{N-1} x^{(N,k)}\right] < C_N\right)$$

où $x^{(N,k)}$ représente le nombre de clients de classe k ($k = 1..N-1$) dans la file N ; $x^{(N,k)}$ correspond à l'état de l'automate $\mathcal{A}^{(N,k)}$.

Ainsi, l'événement s_i ($i = 1..N-1$) concerne les automates $\mathcal{A}^{(i)}$ et $\mathcal{A}^{(N,i)}$, et son taux de franchissement est $\mu_i f$.

Pour exprimer la priorité des clients de différentes classes lors du service par la file N, nous disposons également de $N-2$ fonctions $g^{(k)}$, pour $k = 2..N-1$.

$$g^{(k)}(\tilde{x}) = \delta\left(\left[\sum_{m=1}^{k-1} x^{(N,m)}\right] = 0\right)$$

$g^{(k)}$ exprime la priorité des clients de classe m, avec $m < k$, sur ceux de classe k. Elle vaut 0 quand un client de classe m est présent dans la file N, pour empêcher dans ce cas le service d'un client de classe k. Sa valeur est 1 dans les autres cas.

Les départs de clients sont alors modélisés par des événements locaux d_i ($i = 1..N-1$). L'événement d_i représente le départ d'un client de classe i depuis la file N. d_1 a un taux μ_N^1, et pour $i = 2..N-1$, d_i a un taux $\mu_N^i g^{(i)}$.

La figure 2.12 présente ce réseau d'automates stochastiques pour les valeurs numériques $N = 3$, $C_1 = C_2 = 1$ et $C_3 = 2$.

Ce modèle utilise ainsi des événements synchronisant avec des matrices de synchronisation fonctionnelles.

L'espace d'états accessibles S est de taille

$$|S| = \left(\prod_{i=1}^{N-1}(C_i+1)\right) \times (C_N+1) \times (C_N+2) \times \ldots \times (C_N+(N-1))/(N-1)!$$

et l'espace produit $\hat{S}$ est de taille

$$|\hat{S}| = \left(\prod_{i=1}^{N-1}(C_i+1)\right) \times (C_N+1)^{N-1}$$

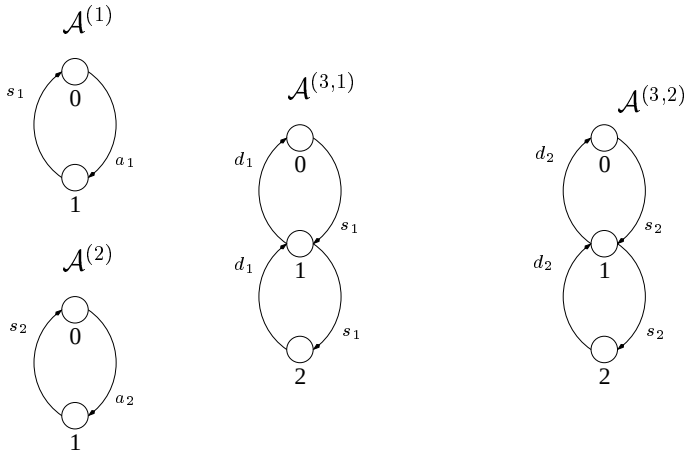


FIG. 2.12 – Réseau de files d'attente, modèle avec fonction et synchronisation – QN2

Ainsi, pour un nombre de files fixé à N, plus C_N est grand, plus la proportion d'états accessibles est faible. Pour C_N tendant vers l'infini, $|\hat{S}| = (N-1)! \times |S|$.

IV.2.2. QN2

Ce deuxième réseau de files d'attente est constitué de quatre files et deux classes de clients. Il est schématisé dans la figure 2.13.

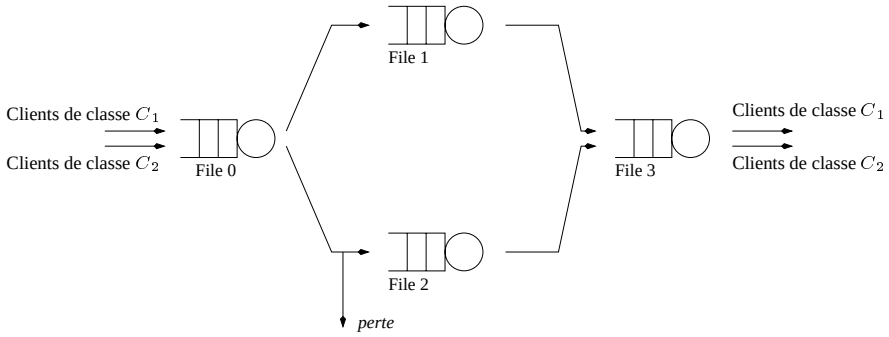


FIG. 2.13 – Réseau de files d'attente – QN2

Les files sont de capacité finie, nous notons K_i la capacité de la file i, pour $i \in [0..3]$. Les clients des deux classes C_1 et C_2 entrent dans le système par la file 0 avec des taux respectifs λ_1 et λ_2. Une fois servis dans cette file, les clients de classe C_1 vont dans la file 1, tandis que ceux de classe C_2 se dirigent vers la file 2.

Si la file 1 est pleine, les clients de classe C_1 sont bloqués dans la file 0. En revanche, les clients de classe C_2 sont perdus (ils quittent le système) si la file 2 est pleine.

Une fois servis respectivement par les files 1 ou 2, les clients C_1 et C_2 vont dans la file 3

si elle n'est pas pleine (sinon, ils sont bloqués dans leur file respective). La file 3 traite les clients des deux classes, et dans cette file les clients C_1 sont prioritaires sur les clients C_2. Ainsi, les clients C_2 ne sont servis que s'il n'y a pas de clients C_1 dans la file.

Au niveau des taux de service, les clients C_j ($j \in [1..2]$) sont servis par le serveur 0 avec un taux inversement proportionnel au nombre de clients C_j présents dans la file 3. La file j ($j \in [1..2]$) ne traite que les clients C_j, qui sont servis avec un taux μ_{jj}. Enfin, le taux de service des clients C_j ($j \in [1..2]$) par le serveur 3 est μ_{3j}.

Modélisation du système QN2 avec fonction

Ce système peut être décomposé en sous-systèmes. Pour chacune des files, nous définissons un automate par classe de clients. Ainsi, pour $i \in [0..3]$ et $j \in [1..2]$, l'automate $\mathcal{A}^{(ij)}$ modélise les clients C_j dans la file i. L'état de cet automate représente le nombre de clients C_j présents dans la file i.

Les arrivées et départs du système sont représentés par des événements locaux car ils n'affectent qu'un seul automate. En revanche, le routage des clients entre les différentes files est modélisé par le biais d'événements synchronisant, vu que les états de deux automates doivent changer simultanément. Ainsi, l'événement e_{ij} ($i \in [0..2]$, $j \in [1..3]$) représente le départ d'un client de la file i et son arrivée dans la file j.

Le comportement de perte des clients C_2 est également exprimé par un événement synchronisant bien qu'une perte change uniquement l'état de l'automate modélisant les clients C_2 dans la file 0 (automate $\mathcal{A}^{(02)}$). Cependant, la perte n'a lieu que lorsque la file 2 est pleine (l'automate $\mathcal{A}^{(22)}$ est dans son dernier état). Nous représentons donc la perte d'un client par un événement qui synchronise le départ d'un client C_2 depuis la file 0 dans l'automate $\mathcal{A}^{(02)}$ avec une transition qui boucle sur elle-même sur le dernier état de l'automate $\mathcal{A}^{(22)}$.

Des taux fonctionnels sont également utilisés, à la fois pour modéliser la limite de capacité des files 0 et 3, le taux de service fonctionnel du serveur 0, et les priorités dans la file 3.

- La fonction f_i représente la limite de capacité de la file i, pour $i = 0$ ou $i = 3$. Elle est évaluée à 1 s'il y a de la place pour un client supplémentaire dans la file i (le nombre de clients C_1 additionné au nombre de clients C_2 est strictement inférieur à la capacité de la file K_i). Sinon, la fonction est évaluée à 0.
  Soit x un état global du SAN, $x = (x^{(01)}, x^{(02)}, x^{(11)}, x^{(22)}, x^{(31)}, x^{(32)})$.
  $x^{(ij)} \in [0..K_i]$ représente l'état de l'automate $\mathcal{A}^{(ij)}$, c'est le nombre de clients de classe C_j présents dans la file i. Alors

$$f_i(x) = \delta\left((x^{(i1)} + x^{(i2)}) < K_i\right)$$

- Le taux de service des clients C_j ($j \in [1..2]$) dans la file 0 est représenté par une fonction g_j. Cette fonction est inversement proportionnelle au nombre de clients C_j dans la file 3 (c'est l'état de l'automate $\mathcal{A}^{(3j)}$). Alors

$$g_j(x) = \frac{\mu_{0j}}{1 + x^{(3j)}}$$

– La dernière fonction représente la priorité des clients C_1 sur les clients C_2 dans la file 3.

$$h_3(x) = \delta\left((x^{(31)} = 0\right)$$

Ce modèle, appelé **QN2** (*Queueing Network*), est illustré dans la figure 2.14.

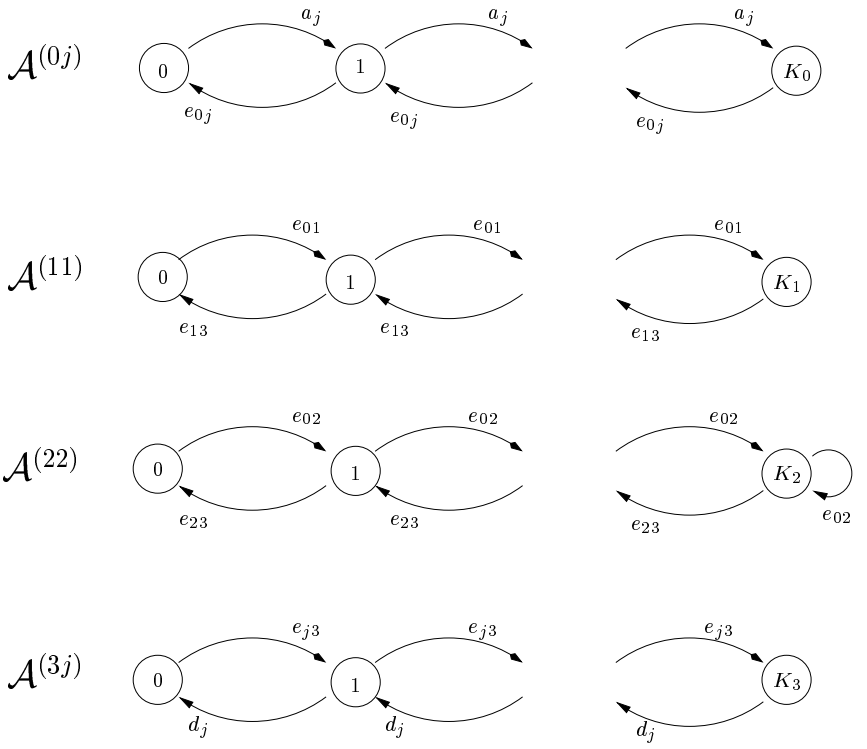


FIG. 2.14 – Réseau de files d'attente, modèle avec fonction – QN2

Les différents événements sont détaillés ci-dessous, pour $j \in [1, 2]$:

– a_j : événement local représentant l'arrivée d'un client de classe C_j dans la file 0. Son taux est $\lambda_j f_0$.
– e_{0j} est un événement synchronisant le départ d'un client C_j de la file 0, et son arrivée dans la file j. Son taux est g_j.
– e_{j3} est un événement synchronisant le départ d'un client C_j de la file j, et son arrivée dans la file 3. Son taux est $\mu_{jj} f_3$.
– d_j : événement local représentant le départ d'un client de classe C_j de la file 3. d_1 a un taux μ_{31}, et d_2 a un taux $\mu_{32} h_3$.

Ce modèle comprend à la fois des événements synchronisant et des événements locaux, et pour chaque type d'événement, nous avons des taux de franchissement fonctionnels. Le descripteur comprend donc une partie locale (somme tensorielle généralisée) et une partie synchronisante (produits tensoriels généralisés).

La taille de l'espace d'états produit de ce modèle est égale à

$$(K_0 + 1)^2 \times (K_1 + 1) \times (K_2 + 1) \times (K_3 + 1)^2$$

Cependant, tous les états ne sont pas accessibles étant donné que la somme des états des automates représentant une même file (clients C_1 et C_2) ne peut pas être supérieure à la capacité de cette file.

IV.3. FILE D'ATTENTE À TEMPS DISCRET : FD

Nous présentons ici un exemple simple à temps discret constitué d'une unique file d'attente de capacité finie C. La file est modélisée par un unique automate $\mathcal{A}$ comportant $C + 1$ états. L'état i ($i \in [0..C]$) signifie qu'il y a i clients dans la file.

Les arrivées de clients sont modélisées par l'événement local a, et les départs par l'événement local d. En temps discret, il peut y avoir durant une même unité de temps une arrivée et un départ. On n'autorise en revanche pas plusieurs départs ou arrivées dans la même unité de temps.

Il faut décider l'ordre dans lequel les arrivées et le service des clients se fait [50] :

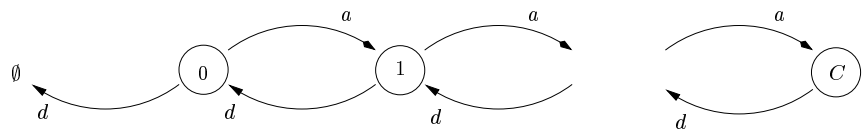
- **Early arrival system** (départs tardifs) – dans ce système, les départs ont lieu après les arrivées. Ainsi, nous pouvons avoir dans une même unité de temps une arrivée, puis un départ.
- **Late arrival system** (arrivées tardives) – dans ce cas, les départs ont lieu au début de l'unité de temps, et les arrivées à la fin.

Dans le cas de départs tardifs, l'événement a (arrivée) doit donc être plus prioritaire que l'événement d (départ). En revanche, lorsque les arrivées sont tardives, c'est d qui est plus prioritaire que a.

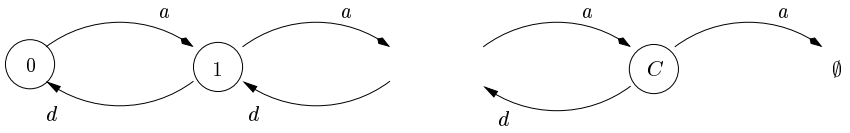
Ce modèle, appelé **FD** (*File d'attente à temps Discret*), est illustré dans la figure 2.15.

Dans le modèle **a.** (départs tardifs), un départ est possible lorsque le système est dans l'état 0 (pas de clients dans la file), mais l'événement n'est pas réalisable dans cet état. En effet, il n'y a pas de clients, il ne peut donc pas y avoir de départ. En revanche, l'événement d est rendu possible à partir de l'état 0 pour autoriser une arrivée et un départ dans la même unité de temps. L'événement a est plus prioritaire que d, ce qui assure les départs tardifs.

Dans le modèle **b.** (arrivées tardives), c'est le phénomène inverse. On autorise une arrivée dans l'état C correspondant à une file d'attente pleine, car l'arrivée peut avoir lieu si un client est servi durant l'unité de temps. Dans ce cas, une place est libérée et un nouveau client peut arriver. Pour ce modèle, ce sont les départs qui sont plus prioritaires que les arrivées, les arrivées sont donc tardives.



a. Early arrival system



b. Late arrival system

FIG. 2.15 – File d'attente à temps discret – FD

V. CONCLUSION

Nous avons présenté ici le formalisme des réseaux d'automates stochastiques, qui est un formalisme de modélisation puissant. En effet, des comportements complexes peuvent être exprimés succinctement grâce à des fonctions et des synchronisations, et les systèmes sont décrits de façon modulaire. Le générateur de la chaîne de Markov sous-jacente peut alors être exprimé comme une formule tensorielle dans le cas du temps continu.

Ce formalisme, déjà bien développé pour les approches à temps continu, est présenté d'une nouvelle manière en mettant l'accent sur la notion d'événements.

Nous proposons également une extension de ce formalisme pour les systèmes à temps discret. Dans ce cas, plusieurs événements peuvent avoir lieu dans la même unité de temps, ce qui représente une différence fondamentale avec les modèles à temps continu. Nous proposons un algorithme pour générer la chaîne de Markov sous-jacente au modèle à temps discret.

Nous avons également présenté des exemples de modélisation à l'aide de réseaux d'automates stochastiques pour donner un aperçu de leur puissance de modélisation. Certains exemples seront exploités par la suite pour donner des résultats numériques sur les nouvelles méthodes développées.

Dans le but de traiter toujours de plus grands modèles, la notion de réseaux d'automates stochastiques avec réplication est introduite dans le chapitre suivant.

CHAPITRE 3

Réseaux d'automates stochastiques avec réplication

Dans ce chapitre, on se place dans le cadre des SANs à temps continu. Une extension pour traiter également le cas des SANs à temps discret pourra être envisagée.

Les réseaux d'automates stochastiques, introduits dans [81, 41] et présentés dans le chapitre précédent, sont un formalisme qui permet de décrire des modèles markoviens de façon efficace, car leur stockage est basé sur une formule tensorielle.

Lors de l'analyse de grandes chaînes de Markov, l'utilisation du formalisme SAN n'est pas toujours suffisant pour pallier au problème de l'explosion du nombre d'états. Les méthodes numériques classiques prennent une grande quantité de mémoire, et les durées d'exécution sont souvent trop longues. Des méthodes itératives d'agrégation/désagrégation (a/d) ont été proposées dans [89] pour l'analyse de grandes chaînes de Markov avec des blocs faiblement couplés, et un algorithme a/d pour les SANs est présenté dans [16].

Agrégation sur l'espace d'états de la chaîne de Markov

Pour surmonter le problème de l'explosion du nombre d'états, on cherche à réduire la complexité de la chaîne de Markov étudiée. Heureusement, de nombreux systèmes contiennent un grand nombre de composants identiques. On peut alors exploiter ces réplications de composants pour générer une chaîne de Markov réduite, en effectuant une agrégation exacte [83, 64, 65, 95]. Des études précédentes sur les conditions d'agrégation faible [14, 71, 72] ont également montré comment grouper des états identiques, mais ce type d'agrégation dépend de l'état initial du modèle. Les conditions d'agrégation et les relations d'équivalence ont été définies et discutées dans [15, 18, 55, 56, 86, 87].

Modèles hiérarchiques

Dans les approches du paragraphe précédent, c'est une agrégation sur l'espace d'état de la chaîne de Markov qui est décrite. Ainsi, la chaîne de Markov agrégée est définie à un niveau bas (sans tenir compte du formalisme de haut niveau employé).

D'autres techniques pour exploiter les réplications sont basées sur des modèles hiérarchiques [12, 76, 75]. De tels modèles sont très utiles pour l'analyse de systèmes complexes, et des techniques pour générer la chaîne de Markov agrégée directement à partir de la spécification de haut niveau ont été développées.

Des algorithmes d'agrégation/désagrégation peuvent également être appliqués sur des modèles hiérarchiques [17]. Une autre approche, basée sur les réseaux de Pétri stochastiques généralisés colorés hiérarchiques [11], permet une génération automatique de la chaîne de Markov agrégée, et également des techniques d'agrégation approchées.

Si l'on considère d'autres formalismes de modélisation, on peut également générer une chaîne de Markov agrégée à partir de réseaux d'activité stochastiques structurés hiérarchiques [85]. Des techniques similaires existent également pour les machines à états finis, comme cela a été montré dans [23].

Réseaux d'automates stochastiques

Toutes ces approches présentent des conditions d'agrégation pour différents formalismes de modélisation, mais aucune propose un moyen satisfaisant d'agréger des réseaux d'automates stochastiques possédant des réplications. Ce chapitre vise donc à présenter une technique équivalente qui peut être utilisée pour agréger de façon efficace des modèles SANs. Les conditions d'agrégation, démontrées précédemment dans [83, 64, 65], seront décrites et utilisées pour prouver qu'un réseau d'automates stochastiques possédant des répliquas peut être fortement agrégé. Les automates identiques du modèle sont identifiés et ils sont alors groupés pour générer une chaîne de Markov agrégée.

Une approche semblable a été décrite par Siegle dans [87], mais elle ne tient pas compte des fonctions, alors que c'est une des caractéristiques essentielles des réseaux d'automates stochastiques. De plus, les travaux de Siegle ne traitent pas des SANs avec plusieurs répliquas comme nous le faisons. Enfin, nous proposons une forme tensorielle de la matrice de la chaîne de Markov agrégée, ce qui n'a encore jamais été fait à notre connaissance.

Plan du chapitre

Nous commençons par introduire la notion de SANs avec réplication, et notamment des SANs avec plusieurs répliquas. Ces notions sont illustrées sur de petits exemples d'étude.

Nous discutons ensuite dans la section II l'agrégation de SANs avec réplication. Des conditions d'agrégation sont détaillées, et nous précisons comment exprimer la matrice de la chaîne de Markov agrégée sous forme tensorielle.

Enfin, la section III illustre les bénéfices de l'agrégation de réseaux d'automates stochastiques à travers quelques résultats théoriques et numériques.

I. DÉFINITION DES SANS AVEC RÉPLICATION

Les grands systèmes contiennent souvent des composants identiques, et la plupart du temps un automate est répliqué plusieurs fois dans le modèle. Nous définissons tout d'abord les *SANs composés d'un seul répliqua*, puis les *SANs avec plusieurs répliquas*. Enfin, nous donnons des éléments sur la façon de détecter les répliquas dans un modèle.

I.1. SANS COMPOSÉS D'UN SEUL RÉPLIQUA

Nous décrivons tout d'abord le concept de SAN composé d'un seul répliqua de façon informelle. Nous donnons ensuite une définition formelle, et illustrons ce concept à travers un exemple.

I.1.1. DESCRIPTION INFORMELLE

De façon informelle, un SAN composé d'un seul répliqua est constitué d'un seul automate répliqué N fois. Cela signifie que les états de chaque automate sont identiques (pour tout $i \in [1..N]$, $S^{(i)} = S^{(1)}$), et les transitions sont répliquées entre chaque couple d'automates (définition 17).

Définition 17 *Soit* $i, j \in [1..N]$, $t^{(i)} \in T^{(i)}$ *et* $t^{(j)} \in T^{(j)}$.
Les transitions $t^{(i)}$ **et** $t^{(j)}$ **sont répliquées entre les automates** $\mathcal{A}^{(i)}$ **et** $\mathcal{A}^{(j)}$ *si :*

- $x_{t^{(i)},dep} = x_{t^{(j)},dep}$ *et* $x_{t^{(i)},arr} = x_{t^{(j)},arr}$
- $\pi_{t^{(i)}} = \pi_{t^{(j)}}$
- *si* $evt_{t^{(i)}}$ *est un événement synchronisant,* $evt_{t^{(i)}} = evt_{t^{(j)}}$
- *si* $evt_{t^{(i)}}$ *est un événement local,* $O_{evt_{t^{(i)}}} = \mathcal{A}^{(i)}$, $O_{evt_{t^{(j)}}} = \mathcal{A}^{(j)}$ *et* $\lambda_{evt_{t^{(i)}}} = \lambda_{evt_{t^{(j)}}}$

Ainsi, pour une transition répliquée donnée, soit elle synchronise tous les automates du SAN, soit l'événement local associé à la transition a le même taux de franchissement dans chaque automate.

De plus, le SAN est composé d'un seul répliqua si et seulement si les fonctions ne sont pas modifiées par permutation de leurs arguments. Par exemple, si $N = 2$, pour tout taux fonctionnel λ et pour tout $x^{(1)}, x^{(2)} \in S^{(1)}$, on doit avoir $\lambda(x^{(1)}, x^{(2)}) = \lambda(x^{(2)}, x^{(1)})$.

I.1.2. DÉFINITION FORMELLE

Nous définissons maintenant de façon formelle un SAN composé d'un seul répliqua :

Définition 18 Un SAN composé d'un seul répliqua *est un SAN avec N automates, tel que, pour $i = 1..N$, on ait :*

- *toutes les matrices locales $Q_l^{(i)}$ sont identiques (égales à Q_l) ;*
- *pour chaque événement synchronisant $e \in \mathcal{ES}$,*
  - *pour chaque automate esclave de l'événement $\mathcal{A}^{(i)} \in O_e$, $i \neq \eta_e$, toutes les matrices $Q_{e^+}^{(i)}$ et $Q_{e^-}^{(i)}$ sont identiques et respectivement égales à Q_{e^+} et Q_{e^-} ;*
  - *pour l'automate maître $\mathcal{A}^{(\eta_e)}$, les matrices $Q_{e^+}^{(\eta_e)}$ et $Q_{e^-}^{(\eta_e)}$ sont respectivement égales à $\lambda_e Q_{e^+}$ et $\lambda_e Q_{e^-}$.*
- *pour chaque événement $e \in \mathcal{E}$, pour toute permutation σ de $[1..N]$, et pour chaque état global $x = (x^{(1)}, ..., x^{(N)})$, $\lambda_e(x) = \lambda_e(\sigma(x))$, où $\sigma(x) = (x^{\sigma(1)}, ..., x^{\sigma(N)})$ (les fonctions ne sont pas changées par permutation des paramètres).*

I.1.3. EXEMPLE DE SAN COMPOSÉ D'UN SEUL RÉPLIQUA

Reprenons l'exemple de partage de ressources RS1, modélisé avec des fonctions, qui a été présenté dans la section IV.1.1 du chapitre 2.

On considère ici que les taux de franchissement des événements de prise et de relâche de ressource sont identiques pour chaque processus :

$$\lambda^{(1)} = ... = \lambda^{(N_p)} = \lambda \quad \text{et} \quad \mu^{(1)} = ... = \mu^{(N_p)} = \mu$$

Alors, pour chaque processus $i \in [1..N_p]$, la matrice de transition locale

$$Q_l^{(i)} = \begin{pmatrix} -\lambda f & \lambda f \\ \mu & -\mu \end{pmatrix}$$

Toutes les matrices sont identiques, et la fonction f n'est pas changée par permutation des paramètres à cause de la commutativité de la somme.

Ce modèle est donc un SAN composé d'un seul répliqua, on l'appelle **RS1-1**.

I.2. SANS AVEC PLUSIEURS RÉPLIQUAS

Nous nous intéressons maintenant à des SANs composés de K automates différents, chacun pouvant être répliqué un certain nombre de fois. On partitionne ainsi l'ensemble des automates, chaque groupe d'automates correspondant à un même automate répliqué.

Nous commençons par introduire quelques définitions et notations concernant cette partition de l'ensemble des automates. Ensuite, nous définissons formellement la notion de SAN avec plusieurs répliquas. Enfin, un exemple illustre ce concept.

I.2.1. PARTITION DES AUTOMATES

Considérons maintenant que nous avons N automates dans le SAN, avec une partition $\mathcal{G}$ telle qu'il y ait K groupes d'automates ($K \in [1..N]$). Nous définissons un ensemble d'indices $(k_g)_{g\in[0..K]}$ délimitant les groupes d'automates :

- $k_0 = 0$;
- pour $g \in [1..K]$, $k_g \in [1..N]$ et $k_{g-1} < k_g$;
- $k_K = N$.

Alors, pour $g \in [1..K]$, le groupe g est constitué des $k_g - k_{g-1}$ automates $\mathcal{A}^{(k_{g-1}+1)}$ à $\mathcal{A}^{(k_g)}$.

Pour $g \in [1..K]$, on introduit les notations suivantes :

- $EI_g = (k_{g-1} + 1, ..., k_g)$ est l'ensemble des indices des automates dans le groupe g ;
- $EA_g = (\mathcal{A}^{(h)})_{h\in EI_g}$ est l'ensemble des automates du groupe g ;
- $R_g = k_g - k_{g-1}$ est le nombre d'automates dans le groupe g, $R_g = |EA_g|$.

Un exemple de partition est illustré dans la figure 3.1.

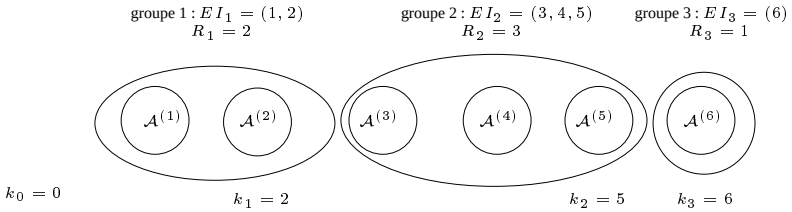


FIG. 3.1 – Partition d'un SAN avec $N = 6$ et $K = 3$

On définit alors un ensemble de permutations tel qu'il puisse y avoir un échange d'automates au sein de chaque groupe, mais pas entre deux groupes distincts.

Définition 19 *Pour un SAN donné et une partition $\mathcal{G}$ (comme définie précédemment), $\mathcal{P}$* **est l'ensemble des permutations** *de $[1..N]$ telles que*
$\sigma \in \mathcal{P} \iff \forall g \in [1..K],\ \forall i \in EI_g,\ \sigma(i) \in EI_g$.

Pour $g \in [1..K]$, soit $\mathcal{P}_g$ l'ensemble des permutations de EI_g (permutations à l'intérieur du groupe g). $\sigma \in \mathcal{P}$ peut être exprimé comme une combinaison des permutations $\sigma_g \in \mathcal{P}_g$, notée par $\sigma = (\sigma_1 \ ... \ \sigma_K)$.

Structure des vecteurs d'état liée à la présence de répliquas : *Soit $x = (x^{(1)}, \ldots, x^{(N)})$ un état global du SAN ; c'est un vecteur d'états locaux. Il peut être décomposé en sous-vecteurs, $x = (x_1 \ ... \ x_K)$ où x_g ($g \in [1..K]$) est un vecteur $x_g = (x^{(h)})_{h \in EI_g}$.*

Alors, pour $\sigma \in \mathcal{P}$, on note $\sigma(x) = (x^{\sigma(1)}, \ldots, x^{\sigma(N)})$.
Si $\sigma = (\sigma_1 \ ... \ \sigma_K)$ comme défini précédemment, alors $\sigma(x) = (\sigma_1(x_1) \ ... \ \sigma_K(x_K))$, et pour $(g \in [1..K])$, $\sigma_g(x_g) = (x^{\sigma(h)})_{h \in EI_g}$.

Noter que la composition de permutations est stable sur $\mathcal{P}$, à savoir : soit $\varphi, \tau \in \mathcal{P}$, alors $\sigma = \varphi \circ \tau$ est aussi dans $\mathcal{P}$. Il est aussi évident que lorsque σ_g parcourt $\mathcal{P}_g$ pour tous les $g \in [1..K]$, alors σ parcourt $\mathcal{P}$.

Lorsque $K = 1$ (SAN composé d'un seul répliqua), $\mathcal{P}$ est l'ensemble des permutations de $[1..N]$.

Si l'on considère l'exemple de la figure 3.1, et si chaque automate a deux états locaux $S0$ et $S1$, l'état global $x = (\mathbf{S1}, \mathbf{S0}, S0, S0, S1, \mathbf{S0})$ signifie que $\mathcal{A}^{(2)}$, $\mathcal{A}^{(3)}$, $\mathcal{A}^{(4)}$ et $\mathcal{A}^{(6)}$ sont dans l'état local $S0$, alors que $\mathcal{A}^{(1)}$ et $\mathcal{A}^{(5)}$ sont dans l'état local $S1$. Soit $\sigma \in \mathcal{P}$ la permutation telle que $\sigma(x) = (\mathbf{S0}, \mathbf{S1}, S0, S1, S0, \mathbf{S0})$. Dans chaque état global (x et $\sigma(x)$), un automate du premier groupe, deux du deuxième et un du troisième sont dans l'état $S0$; un automate du premier groupe, un du deuxième et aucun du troisième sont dans l'état $S1$.

Le fait d'avoir le même nombre d'automates dans chaque état local signifie que l'on peut passer d'un état global à un autre par une permutation de $\mathcal{P}$.

I.2.2. DÉFINITION FORMELLE

Nous définissons maintenant de façon formelle un SAN avec plusieurs répliquas :

Définition 20 Un SAN avec plusieurs répliquas *est un SAN avec N automates et une partition de l'ensemble des automates $\mathcal{G}$, tel que, pour $g = 1..K$, on ait :*

- *pour tout $i \in EI_g$, les matrices locales $Q_l^{(i)}$ sont identiques (égales à $Q^{(l, groupe\ g)}$) ;*
- *pour chaque événement synchronisant $e \in \mathcal{ES}$,*
  - *pour chaque automate esclave de l'événement dans le groupe g, $\mathcal{A}^{(i)} \in O_e \cap EA_g$, $i \neq \eta_e$, toutes les matrices $Q_{e^+}^{(i)}$ et $Q_{e^-}^{(i)}$ sont identiques et respectivement égales à $Q^{(e^+, groupe\ g)}$ et $Q^{(e^-, groupe\ g)}$;*
  - *pour l'automate maître $\mathcal{A}^{(\eta_e)}$, si $\eta_e \in EI_g$ (il se peut que l'automate maître soit dans un autre groupe), les matrices $Q_{e^+}^{(\eta_e)}$ et $Q_{e^-}^{(\eta_e)}$ sont respectivement égales à $\lambda_e Q^{(e^+, groupe\ g)}$ et $\lambda_e Q^{(e^-, groupe\ g)}$.*
- *pour chaque événement $e \in \mathcal{E}$, pour toute permutation $\sigma \in \mathcal{P}$, et pour chaque état global x, $\lambda_e(x) = \lambda_e(\sigma(x))$ (pour chaque groupe g, les fonctions ne sont pas changées par permutation des paramètres "état de $\mathcal{A}^{(k_{g-1}+1)}$" à "état de $\mathcal{A}^{(k_g)}$").*

On peut remarquer qu'un SAN est dit **sans répliqua** s'il y a N groupes avec un seul automate dans chaque groupe ($K = N$ et $k_g = g$ pour $g \in [1..N]$). Un tel exemple était décrit dans la figure 2.4 (cf chapitre 2).

D'un autre coté, un **SAN composé d'un seul répliqua** est un SAN pour lequel tous les automates sont dans le même groupe ($K = 1$), comme pour l'exemple *RS1-1* présenté en section I.1.3.

Un **SAN avec plusieurs répliquas** est donc un SAN pour lequel $1 < K < N$.

Il faut également remarquer que les événements synchronisant peuvent impliquer des automates dans plusieurs groupes (ou bien dans un seul groupe). Les conditions exprimées dans la définition 20 expriment le fait que les automates d'un même groupe ont le même comportement pour chaque événement synchronisant.

I.2.3. EXEMPLE DE SAN AVEC PLUSIEURS RÉPLIQUAS

Reprenons l'exemple de partage de ressources RS1, modélisé avec des fonctions, qui a été présenté dans la section IV.1.1 du chapitre 2.

On considère ici qu'il y a K groupes de processus ayant des taux de franchissement différents pour la prise et la relâche de ressource. Les N_r ressources peuvent en revanche être utilisées indifféremment par n'importe quel processus de n'importe quel groupe. Pour $g \in [1..K]$, soit λ_g le taux de prise de ressource, et μ_g le taux de relâche de ressource par un processus du groupe g. On a donc :

$$\forall i \in EI_g \quad \lambda^{(i)} = \lambda_g \text{ et } \mu^{(i)} = \mu_g$$

Ce modèle ne contient pas d'événements synchronisant, et pour chaque groupe $g \in [1..K]$ et chaque processus $i \in EI_g$ dans ce groupe, la matrice de transition locale est

$$Q_l^{(i)} = \begin{pmatrix} -\lambda_g f & \lambda_g f \\ \mu_g & -\mu_g \end{pmatrix}$$

A l'intérieur de chaque groupe, les matrices de transition locales sont identiques. De plus, la fonction f n'est pas changée par permutation des paramètres.

Ce modèle est donc un SAN avec plusieurs répliquas, on l'appelle **RS1-K**.

I.3. COMMENT DÉTECTE-T-ON LES RÉPLIQUAS ?

La partition $\mathcal{G}$ des automates répliqués est habituellement donnée par l'utilisateur lors de la phase de modélisation du système étudié. Les grands systèmes sont en effet souvent décrits par un ensemble de composants identiques. Lors de la modélisation en tant que réseau d'automates stochastiques, on doit juste s'assurer que les propriétés de *SAN avec plusieurs répliquas* sont vérifiées pour la partition $\mathcal{G}$ donnée. Ainsi, nous n'avons pas besoin de détecter les répliquas, vu qu'ils sont donnés par le modèle lui-même.

II. AGRÉGATION DE SANS AVEC RÉPLICATION

Maintenant que nous avons introduit la notion de SANs avec réplication, on peut procéder à l'agrégation de tels SANs. Dans ce qui suit, on suppose que l'on dispose d'un SAN initial, et d'une partition des automates $\mathcal{G}$. On veut alors obtenir une chaîne de Markov réduite, résultant d'une agrégation forte.

On commence par montrer sur un exemple comment l'agrégation fonctionne, de façon intuitive. Ensuite, on donne les conditions nécessaires pour réaliser une agrégation forte, et on prouve ainsi qu'un SAN avec plusieurs répliquas peut être agrégé. Enfin, nous montrons que la matrice de la chaîne de Markov agrégée peut s'exprimer sous forme tensorielle.

II.1. EXEMPLE D'AGRÉGATION

L'agrégation est basée sur le fait que, grâce aux répliquas, certains états globaux du SAN ont la même probabilité stationnaire. On illustre l'agrégation sur l'exemple du partage de ressources **RS1-1** (cf section I.1.3), avec $N_p = 3$ processus et $N_r = 2$ ressources. Ce SAN est composé d'un seul répliqua. Dans cet exemple, $\mathcal{P}$ est donc l'ensemble des permutations de $[1..N_p]$.

Tous les états du SAN initial équivalents par une permutation de $\mathcal{P}$ sont groupés en un unique état de la chaîne de Markov agrégée. On peut arbitrairement choisir un état particulier pour représenter chaque classe d'équivalence.

Pour notre exemple, on code l'état *repos* par 1 et l'état *actif* par 2, un état global est alors noté $Sijk$, où $i = 1, 2$ est l'état du premier processus, $j = 1, 2$ est l'état du deuxième processus, et $k = 1, 2$ est l'état du troisième processus. La chaîne de Markov agrégée comporte quatre états, que l'on peut représenter par $S111$, $S112$,$S122$ et $S222$.

Chaque état global du SAN initial est équivalent à l'un de ces états. Cela signifie que l'on a juste besoin de savoir **combien** de processus utilisent les ressources, mais peu importe **quels** processus les utilisent. Par exemple, l'état $S112$ signifie qu'une seule ressource est utilisée.

Si $C0$ et $C1$ représentent deux classes d'équivalence différentes (ce sont donc des états de la chaîne de Markov agrégée), le taux de transition de l'état $C0$ vers l'état $C1$ de la chaîne de Markov agrégée s'obtient en faisant la somme des taux de transition depuis un des états de la classe $C0$ vers tous les états de la classe $C1$ dans le SAN initial. On verra par la suite que le choix de l'état de $C0$ dans le SAN initial n'affecte pas le résultat.

La chaîne de Markov agrégée pour l'exemple Mutex1 est représentée dans la figure 3.2, et la définition formelle de l'agrégation de SAN suit.

On peut remarquer que l'état $S222$ n'est pas atteignable. L'espace d'état de la chaîne de Markov agrégée, noté $\hat{S}_{ag}$, comporte souvent des états non atteignables. On peut alors les

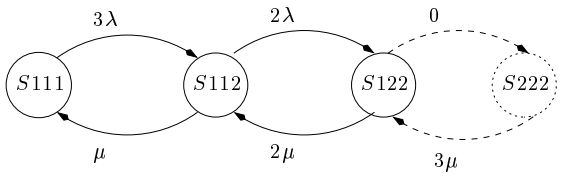


FIG. 3.2 – RS1-1 – $N_p = 3$ et $N_r = 2$ – chaîne de Markov agrégée

supprimer de la chaîne de Markov, et on obtient un nouvel espace d'état noté S_{ag}.

II.2. CONDITIONS D'AGRÉGATION

Maintenant que nous avons défini les répliquas dans les SANs, nous nous intéressons aux conditions d'agrégation forte de tels SANs. On considère une partition de l'espace d'état $\Omega = (\Omega_1, ..., \Omega_\Gamma)$, et on rappelle la définition de l'agrégation forte ([64, 65, 83, 95]).

Définition 21 Une chaîne de Markov peut être agrégée fortement *sur la partition Ω si pour tout vecteur initial, la chaîne agrégée (dont les états sont Ω_γ, pour $\gamma \in [1..\Gamma]$) est une chaîne de Markov, et si les taux de transition de cette chaîne ne dépendent pas du vecteur initial.*

Ensuite, une condition d'agrégation forte est donnée dans le théorème suivant [83, 64, 65, 95] :

> **Théorème de Rosenblatt :** *On considère une chaîne de Markov à temps continu avec un espace d'état Ω, et une partition de l'espace d'état $\Omega = \bigcup_\gamma \Omega_\gamma$, ($\gamma \in [1..\Gamma]$). Si, pour tout $\beta, \gamma \in [1..\Gamma]$, la probabilité de passer d'un état $x \in \Omega_\beta$ à Ω_γ a toujours la même valeur quel que soit l'état x de Ω_β, alors la chaîne de Markov peut être agrégée fortement sur la partition Ω.*

Une preuve de ce théorème se trouve dans [64].

Dans la chaîne de Markov agrégée, le taux de transition de l'état Ω_β vers Ω_γ ($\beta, \gamma \in [1..\Gamma]$) est la somme des taux de transition dans le SAN initial pour aller de l'un des états de Ω_β vers chaque état de Ω_γ. Grâce au théorème de Rosenblatt, on peut arbitrairement choisir n'importe quel état de Ω_β pour calculer ces taux de transition, le résultat obtenu sera toujours le même.

Maintenant que nous avons vu les conditions qu'il faut réunir pour pouvoir agréger fortement une chaîne de Markov, nous nous intéressons à l'agrégation des SANs avec plusieurs répliquas, et prouvons qu'**un SAN avec plusieurs répliquas peut être agrégé fortement**.

Pour cela, on commence par définir la partition de l'espace d'état que l'on désire utiliser dans le théorème de Rosenblatt, puis on prouve que la condition de Rosenblatt est vérifiée sur la partition considérée (Lemme 1).

Définition 22 *Deux états globaux x et y ($x, y \in \hat{S}$) sont* équivalents *si :*
$\exists \sigma \in \mathcal{P} \;\; \sigma(x) = y$
On définit la **partition de l'espace d'état** $\Omega = (\Omega_1, ..., \Omega_\Gamma)$*, où chaque* Ω_γ *($\gamma \in [1..\Gamma]$) correspond à un ensemble d'états équivalents. Ce sont des classes d'équivalence. Alors on peut choisir pour chaque* Ω_γ *un état particulier* $r\gamma \in \Omega_\gamma$*, et pour chaque état* $x \in \Omega_\gamma$*, il existe une permutation* $\tau \in \mathcal{P}$ *telle que* $\tau(x) = r\gamma$*.*

Ainsi, si deux états globaux sont équivalents, il y a dans chaque groupe le même nombre d'automates dans un état local donné pour les deux états.

On peut remarquer que si $x = (x^{(1)}, ..., x^{(N)})$ est un état global du SAN, alors pour $\tau \in \mathcal{P}$, $\tau(x)^{(i)} = x^{(\tau(i))}$.

Le descripteur du SAN Q décrit dans la formule 2.7 peut être décomposé en trois parties :

$$Q = L + \sum_{e \in \mathcal{ES}} (Pe + Ne) \tag{3.1}$$

✍ **Soit**

- Q : le descripteur du SAN ;
- L : la matrice correspondant à la partie locale du descripteur ;
- Pe : la matrice correspondant à la partie synchronisante positive du descripteur, pour $e \in \mathcal{ES}$;
- Ne : la matrice correspondant à la partie synchronisante négative du descripteur, pour $e \in \mathcal{ES}$;
- q_{xy} : l'élément dans la ligne x et la colonne y de la matrice Q ; c'est la probabilité de passer de l'état global x à l'état global y dans le SAN : $q_{xy} = l_{xy} + \sum_{e \in \mathcal{ES}} (pe_{xy} + ne_{xy})$;
- $q_{x\Omega_\gamma}$: la probabilité de passer de l'état global x à l'un des états de Ω_γ ($\gamma \in [1..\Gamma]$) : $q_{x\Omega_\gamma} = \sum_{y \in \Omega_\gamma} q_{xy}$.

La notation $q_{x\Omega_\gamma}$ peut être étendue aux matrices L, Pe et Ne ($e \in \mathcal{ES}$) :

$$l_{x\Omega_\gamma} = \sum_{y \in \Omega_\gamma} l_{xy}, \qquad pe_{x\Omega_\gamma} = \sum_{y \in \Omega_\gamma} pe_{xy}, \qquad ne_{x\Omega_\gamma} = \sum_{y \in \Omega_\gamma} ne_{xy}$$

On peut remarquer en outre que tous les états de Ω_γ sont par définition équivalents par une permutation de $\mathcal{P}$, d'où

$$l_{x\Omega_\gamma} = \sum_{y \in \Omega_\gamma} l_{xy} = \sum_{\sigma \in \mathcal{P}} l_{x\sigma(r\gamma)}$$

On peut naturellement écrire la même chose pour pe et ne.

Pour prouver qu'un SAN avec plusieurs répliquas peut être agrégé fortement, il reste à démontrer le lemme suivant :

Lemme 1 : *La condition de Rosenblatt est vérifiée pour le SAN sur la partition* $\Omega = (\Omega_1, ..., \Omega_\Gamma)$ *(cf définition 22).*
En d'autres termes, pour chaque $\beta, \gamma \in [1..\Gamma]$*, la probabilité de passer d'un état* $x \in \Omega_\beta$ *à* Ω_γ *a la même valeur pour tout état* x *de* Ω_β *:*
$\forall x \in \Omega_\beta \;\; q_{x\Omega_\gamma} = q_{r\beta\Omega_\gamma}$.

Pour démontrer le Lemme 1, on décompose le problème en deux parties, qui correspondent respectivement à la partie locale du descripteur (Lemme 2) et à la partie synchronisante (Lemme 3).

Lemme 2 : *Pour chaque* $\beta, \gamma \in [1..\Gamma]$*, la probabilité de passer d'un état* $x \in \Omega_\beta$ *à* Ω_γ *par un événement local a la même valeur pour tout état* x *de* Ω_β *:*
$\forall x \in \Omega_\beta \;\; l_{x\Omega_\gamma} = l_{r\beta\Omega_\gamma}$.

• DÉMONSTRATION DU LEMME 2

Soit $\beta, \gamma \in [1..\Gamma]$, $x \in \Omega_\beta$ et soit $\tau \in \mathcal{P}$ la permutation telle que $x = \tau(r\beta)$.

Avec les définitions d'algèbre tensorielle généralisée (cf chapitre 2, section I page 23), pour $y \in \Omega_\gamma$, on a

$$l_{xy} = \sum_{i=1}^{N} \left[q^{(i)}_{l\, x^{(i)} y^{(i)}}(x) \prod_{j=1, j\neq i}^{N} \delta_{x^{(j)} y^{(j)}} \right]$$

où $Q_l^{(i)}$ est la matrice de transition locale de l'automate $\mathcal{A}^{(i)}$, et $\delta_{uv} = 1$ si $u = v$, et 0 sinon.

Pour simplifier les notations, on note par la suite

$$\Delta_{x^{(i)} y^{(i)}} = \prod_{j=1, j\neq i}^{N} \delta_{x^{(j)} y^{(j)}}$$

Alors on a

$$l_{x\Omega_\gamma} = \sum_{y \in \Omega_\gamma} l_{xy} = \sum_{\sigma \in \mathcal{P}} l_{x\sigma(r\gamma)} = \sum_{\sigma \in \mathcal{P}} \sum_{i=1}^{N} q^{(i)}_{l\, x^{(i)} r\gamma^{(\sigma(i))}}(x) \;\; \Delta_{x^{(i)} r\gamma^{(\sigma(i))}}$$

De plus, $x = \tau(r\beta)$, donc

$$l_{x\Omega_\gamma} = \sum_{\sigma \in \mathcal{P}} \sum_{i=1}^{N} q^{(i)}_{l\, r\beta^{(\tau(i))}\; r\gamma^{(\sigma(i))}}(\tau(r\beta)) \;\; \Delta_{r\beta^{(\tau(i))}\; r\gamma^{(\sigma(i))}}$$

On décompose maintenant cette équation pour chaque groupe du SAN. Toutes les matrices locales $Q_l^{(i)}$ correspondant aux automates d'un groupe $g \in [1..K]$ ($i \in EI_g$) sont identiques, égales à $Q^{(l, groupe\ g)}$.

Pour $\sigma \in \mathcal{P}$, soit $\varphi = \sigma \circ \tau^{-1}$. La composition de permutations est une opération stable sur $\mathcal{P}$, donc $\varphi \in \mathcal{P}$. Alors $\sigma = \varphi \circ \tau$ et lorsque σ parcourt l'ensemble des permutations $\mathcal{P}$,

φ le fait également, dans un ordre différent. Grâce à la commutativité de la somme, on peut donc remplacer $\sum_{\sigma\in\mathcal{P}}$ par $\sum_{\varphi\in\mathcal{P}}$. Alors on a :

$$l_{x\Omega_\gamma} = \sum_{\varphi\in\mathcal{P}} \sum_{g=1}^{K} \sum_{i\in EI_g} q^{(l,groupe\ g)}_{r\beta(\tau(i))\ r\gamma(\varphi\circ\tau(i))}(\tau(r\beta))\ \ \Delta_{r\beta(\tau(i))\ r\gamma(\varphi\circ\tau(i))}$$

Les fonctions ne sont pas changées par des permutations de $\mathcal{P}$, et $\tau \in \mathcal{P}$, donc on peut remplacer $\tau(r\beta)$ par $r\beta$ dans l'équation ci-dessus. De plus, pour chaque groupe g, lorsque i parcourt EI_g, $\tau(i)$ également, mais dans un ordre différent (car c'est une permutation à l'intérieur de chaque groupe, cf définition 19). Grâce à la commutativité de la somme, on peut changer l'ordre et remplacer $\tau(i)$ par i dans l'équation :

$$l_{x\Omega_\gamma} = \sum_{\varphi\in\mathcal{P}} \sum_{g=1}^{K} \sum_{i\in EI_g} q^{(l,groupe\ g)}_{r\beta(i)\ r\gamma(\varphi(i))}(r\beta)\ \ \Delta_{r\beta(i)\ r\gamma(\varphi(i))}$$

$$l_{x\Omega_\gamma} = \sum_{\varphi\in\mathcal{P}} \sum_{i=1}^{N} q^{(i)}_{l\ r\beta(i)\ r\gamma(\varphi(i))}(r\beta)\ \ \Delta_{r\beta(i)\ r\gamma(\varphi(i))}$$

Et finalement,

$$l_{x\Omega_\gamma} = \sum_{\varphi\in\mathcal{P}} l_{r\beta\ \varphi(r\gamma)} = l_{r\beta\ \Omega_\gamma}$$

□

Lemme 3 : *Pour chaque $\beta, \gamma \in [1..\Gamma]$, la probabilité de passer d'un état $x \in \Omega_\beta$ à Ω_γ par un événement synchronisant a la même valeur pour tout état x de Ω_β :*
$\forall x \in \Omega_\beta\ \ \sum_{e\in\mathcal{ES}} \left(pe_{x\Omega_\gamma} + ne_{x\Omega_\gamma}\right) = \sum_{e\in\mathcal{ES}} \left(pe_{r\beta\Omega_\gamma} + ne_{r\beta\Omega_\gamma}\right)$.

• DÉMONSTRATION DU LEMME 3

Soit $e \in \mathcal{ES}$, $\beta, \gamma \in [1..\Gamma]$, $x \in \Omega_\beta$ et soit $\tau \in \mathcal{P}$ la permutation telle que $x = \tau(r\beta)$. On démontre dans un premier temps que $pe_{x\Omega_\gamma} = pe_{r\beta\Omega_\gamma}$. La démonstration pour ne se fait exactement de la même façon.

Avec les définitions d'algèbre tensorielle généralisée (cf chapitre 2, section I page 23), pour $y \in \Omega_\gamma$, on a

$$pe_{xy} = \prod_{i=1}^{N} q^{(i)}_{e+\ x^{(i)}y^{(i)}}(x)$$

où $Q^{(i)}_{e+}$ est la matrice de synchronisation positive de l'automate $\mathcal{A}^{(i)}$.

De plus, $x = \tau(r\beta)$, donc

$$pe_{x\Omega_\gamma} = \sum_{y\in\Omega_\gamma} pe_{xy} = \sum_{\sigma\in\mathcal{P}} pe_{x\sigma(r\gamma)} = \sum_{\sigma\in\mathcal{P}} \prod_{i=1}^{N} q^{(i)}_{e+\ r\beta(\tau(i))\ r\gamma(\sigma(i))}(\tau(r\beta))$$

On décompose maintenant cette équation pour chaque groupe du SAN. Toutes les matrices de synchronisation positives $Q_{e^+}^{(i)}$ correspondant aux automates esclaves de l'événement e dans un groupe $g \in [1..K]$ ($i \in EI_g$, $i \neq \eta_e$) sont identiques, égales à $Q^{(e^+,groupe\ g)}$. L'automate maître de l'événement $\mathcal{A}^{\eta_e}$ est présent dans un des groupes, et sa matrice est égale à $\lambda_e\ Q^{(e^+,groupe\ g)}$ (cf définition 20).

Pour $\sigma \in \mathcal{P}$, soit $\varphi = \sigma \circ \tau^{-1}$. La composition de permutations est une opération stable sur $\mathcal{P}$, donc $\varphi \in \mathcal{P}$. Alors $\sigma = \varphi \circ \tau$ et lorsque σ parcourt l'ensemble des permutations $\mathcal{P}$, φ le fait également, dans un ordre différent. Grâce à la commutativité de la somme, on peut donc remplacer $\sum_{\sigma\in\mathcal{P}}$ par $\sum_{\varphi\in\mathcal{P}}$. Alors on a :

$$pe_{x\Omega_\gamma} = \sum_{\varphi\in\mathcal{P}} \prod_{g=1}^{K} \left(\lambda_e(\tau(r\beta)) \prod_{i\in EI_g} q^{(e+,groupe\ g)}_{r\beta^{(\tau(i))}\ r\gamma^{(\varphi\circ\tau(i))}}(\tau(r\beta)) \right)$$

Les fonctions ne sont pas changées par des permutations de $\mathcal{P}$, et $\tau \in \mathcal{P}$, donc on peut remplacer $\tau(r\beta)$ par $r\beta$ dans l'équation ci-dessus. De plus, pour chaque groupe g, lorsque i parcourt EI_g, $\tau(i)$ également, mais dans un ordre différent (car c'est une permutation à l'intérieur de chaque groupe, cf définition 19). Grâce à la commutativité du produit, on peut changer l'ordre et remplacer $\tau(i)$ par i dans l'équation :

$$pe_{x\Omega_\gamma} = \sum_{\varphi\in\mathcal{P}} \prod_{g=1}^{K} \left(\lambda_e(r\beta) \prod_{i\in EI_g} q^{(e+,groupe\ g)}_{r\beta^{(i)}\ r\gamma^{(\varphi(i))}}(r\beta) \right)$$

$$pe_{x\Omega_\gamma} = \sum_{\varphi\in\mathcal{P}} \prod_{i=1}^{N} q^{(i)}_{e+\ r\beta^{(i)}\ r\gamma^{(\varphi(i))}}(r\beta)$$

Et finalement, $pe_{x\Omega_\gamma} = \sum_{\varphi\in\mathcal{P}} pe_{r\beta\ \varphi(r\gamma)} = pe_{r\beta\ \Omega_\gamma}$.

De façon identique, on peut prouver que $ne_{x\Omega_\gamma} = ne_{r\beta\ \Omega_\gamma}$. Ceci est vrai pour chaque événement $e \in \mathcal{ES}$, donc on a finalement

$$\sum_{e\in\mathcal{ES}} \left(pe_{x\Omega_\gamma} + ne_{x\Omega_\gamma}\right) = \sum_{e\in\mathcal{ES}} \left(pe_{r\beta\Omega_\gamma} + ne_{r\beta\Omega_\gamma}\right)$$

□

• DÉMONSTRATION DU LEMME 1

Soit $\beta, \gamma \in [1..\Gamma]$, $x \in \Omega_\beta$ et soit $\tau \in \mathcal{P}$ la permutation telle que $x = \tau(r\beta)$.

Grâce à la décomposition de Q, on a

$$\begin{aligned} q_{x\Omega_\gamma} &= \textstyle\sum_{y\in\Omega_\gamma} q_{xy} \ = \sum_{y\in\Omega_\gamma} \left(l_{xy} + \sum_{e\in\mathcal{ES}} (pe_{xy} + ne_{xy})\right) \\ &= \textstyle\sum_{y\in\Omega_\gamma} l_{xy} + \sum_{e\in\mathcal{ES}} \left(\sum_{y\in\Omega_\gamma} pe_{xy} + \sum_{y\in\Omega_\gamma} ne_{xy}\right) \\ &= l_{x\Omega_\gamma} + \textstyle\sum_{e\in\mathcal{ES}} \left(pe_{x\Omega_\gamma} + ne_{x\Omega_\gamma}\right) \end{aligned}$$

Et finalement, en appliquant le Lemme 2 et le Lemme 3, on a

$$q_{x\Omega_\gamma} = l_{r\beta\Omega_\gamma} + \sum_{e\in\mathcal{ES}} \left(pe_{r\beta\Omega_\gamma} + ne_{r\beta\Omega_\gamma}\right) = q_{r\beta\Omega_\gamma}$$

□

Le Lemme 1 et l'application du théorème de Rosenblatt nous assurent que la chaîne de Markov sous-jacente au SAN peut être agrégée fortement sur la partition Ω.

Les états de la chaîne de Markov agrégée sont les classes d'équivalence définies par les permutations de $\mathcal{P}$. Ces classes d'équivalence sont construites sur les états de l'espace produit du SAN initial, qui peut donc contenir des états non accessibles. Ainsi, la chaîne de Markov agrégée peut avoir des états non accessibles. On note $\hat{S}_{ag}$ l'espace des classes d'équivalence (espace des états de la chaîne de Markov agrégée), et S_{ag} l'espace des états accessibles.

II.3. EXPRESSION TENSORIELLE DE LA MATRICE DE LA CHAÎNE DE MARKOV AGRÉGÉE

Nous présentons maintenant comment exprimer la matrice de la chaîne de Markov agrégée sous forme tensorielle.

Rappelons que le descripteur Q du SAN initial peut être décomposé en trois parties (formule 3.1) : $Q = L + \sum_{e\in\mathcal{ES}}(P_e + N_e)$. L est une somme tensorielle, et tous les P_e et N_e sont des produits tensoriels.

✍ **Soit**

x	un état de l'espace d'états initial $\hat{S}$;
$\tilde{x}$	un état de l'espace d'états agrégé $\hat{S}_{ag}$, c'est un ensemble d'états équivalents de $\hat{S}$;
$\tilde{Q}$	le générateur de la chaîne de Markov agrégée.

$\tilde{Q}$ est défini par tous ses éléments. Ainsi, pour tout $\tilde{x} \in \hat{S}_{ag}$, $\tilde{y} \in \hat{S}_{ag}$, et pour tout $x \in \tilde{x}$ (état de la classe d'équivalence $\tilde{x}$), on a

$$\tilde{q}_{\tilde{x}\tilde{y}}(\tilde{x}) = \sum_{y\in\tilde{y}} q_{xy}(x) = \sum_{\sigma\in\mathcal{P}} q_{x\sigma(y)}(x)$$

De plus, par définition de Q, $q_{xy} = l_{xy} + \sum_{e\in\mathcal{ES}}(pe_{xy} + ne_{xy})$, et finalement

$$\tilde{q}_{\tilde{x}\tilde{y}}(\tilde{x}) = \sum_{\sigma\in\mathcal{P}} l_{x\sigma(y)}(x) + \sum_{e\in\mathcal{ES}} \left(\sum_{\sigma\in\mathcal{P}} pe_{x\sigma(y)}(x) + \sum_{\sigma\in\mathcal{P}} ne_{x\sigma(y)}(x) \right)$$

Ainsi, la matrice $\tilde{Q}$ peut être exprimée comme la somme de matrices :
$\tilde{Q} = \tilde{L} + \sum_{e \in \mathcal{ES}} (\tilde{P}_e + \tilde{N}_e)$, où, pour chaque état global $\tilde{x}$ et $\tilde{y}$, et pour $a = l$, $a = pe$ ou $a = ne$ ($e \in \mathcal{ES}$),

$$\tilde{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \sum_{\sigma \in \mathcal{P}} a_{x\sigma(y)}(x)$$

Définition 23 *Soit A l'une des matrices L, P_e ou N_e ($e \in \mathcal{ES}$).*
$\hat{A}$ est une **version agrégée** *de la matrice A sous forme d'une expression tensorielle.*

Les matrices $\hat{A}$ seront définies par la suite. Elles sont obtenues intuitivement en effectuant une agrégation séparée de chaque groupe d'automates répliqués, puis en combinant les matrices obtenues par une formule tensorielle. Ces agrégations produisent donc un SAN équivalent, qui n'est pas défini ici. On travaille uniquement sur l'expression du descripteur de ce SAN.

On veut alors prouver que $\tilde{Q} = \hat{L} + \sum_{e \in \mathcal{ES}} (\hat{P}_e + \hat{N}_e)$ ($\tilde{Q}$ est la matrice de la chaîne de Markov agrégée). Ainsi, on aura une expression tensorielle de $\tilde{Q}$.

Pour prouver ceci, on raisonne sur chaque terme a, où a peut être l, pe ou ne ($e \in \mathcal{ES}$), et on montre que, pour tout $\tilde{x} \in \hat{S}_{ag}$, $\tilde{y} \in \hat{S}_{ag}$, et pour tout $x \in \tilde{x}$ (état de la classe d'équivalence $\tilde{x}$),

$$\tilde{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \hat{a}_{\tilde{x}\tilde{y}}(\tilde{x}) \tag{3.2}$$

On commence par raisonner sur les produits tensoriels P_e et N_e, pour $e \in \mathcal{ES}$. Ensuite on étudie le cas de la somme tensorielle L. Enfin, on en déduit une expression tensorielle de $\tilde{Q}$.

II.3.1. PRODUIT TENSORIEL

Soit A l'un des produits tensoriels P_e ou N_e, $e \in \mathcal{ES}$: $A = \bigotimes_{g\ i=1}^{N} Q^{(i)}$.

Soit $x \in \hat{S}$ et $y \in \hat{S}$ deux états globaux. Avec la définition du produit tensoriel généralisé (cf chapitre 2, section I.1 page 24), on a

$$a_{xy}(x) = \prod_{i=1}^{N} q^{(i)}_{x^{(i)}y^{(i)}}(x)$$

Donc

$$\tilde{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \sum_{\sigma \in \mathcal{P}} a_{x\sigma(y)}(x) = \sum_{\sigma \in \mathcal{P}} \prod_{i=1}^{N} q^{(i)}_{x^{(i)}\sigma(y^{(i)})}(x)$$

Or σ peut être exprimée comme une combinaison de permutations (définition 19), $\sigma = (\sigma_1 \ ... \ \sigma_K)$. On a donc

$$\tilde{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \sum_{\sigma_1 \in \mathcal{P}_1} ... \sum_{\sigma_K \in \mathcal{P}_K} \prod_{i=1}^{N} q^{(i)}_{x^{(i)}\sigma(y^{(i)})}(x)$$

Si on décompose le produit suivant chaque groupe, on peut remplacer dans le produit σ par une des permutations σ_g, où $g \in [1..K]$, puis factoriser les termes indépendants :

$$\tilde{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \prod_{h=1}^{K} \left(\sum_{\sigma_h \in \mathcal{P}_h} \prod_{i \in EI_h} q^{(i)}_{x^{(i)}\sigma_h(y^{(i)})}(x) \right) \tag{3.3}$$

On définit alors la matrice agrégée $\hat{A}$ (définition 23) par un produit tensoriel de K matrices :

$$\hat{A} = \bigotimes_{\substack{g \\ h \in [1..K]}} \hat{A}^h$$

et la matrice $\hat{A}^h$, $h \in [1..K]$, est définie par

$$\hat{a}^h_{\tilde{x}_h \tilde{y}_h}(\tilde{x}) = \sum_{\sigma_h \in \mathcal{P}_h} \prod_{i \in EI_h} q^{(i)}_{x^{(i)}\sigma_h(y^{(i)})}(\tilde{x})$$

Alors

$$\hat{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \prod_{h=1}^{K} \hat{a}^h_{\tilde{x}_h \tilde{y}_h} = \prod_{h=1}^{K} \sum_{\sigma_h \in \mathcal{P}_h} \prod_{i \in EI_h} q^{(i)}_{x^{(i)}\sigma_h(y^{(i)})}(\tilde{x}) \tag{3.4}$$

Grâce aux propriétés des fonctions dans le cadre de SANs avec répliquas, on obtient toujours le même résultat pour tout $x \in \tilde{x}$. Ainsi, les formules 3.3 et 3.4 sont identiques, ce qui prouve la formule 3.2 ($\tilde{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \hat{a}_{\tilde{x}\tilde{y}}(\tilde{x})$) dans le cas où A est un produit tensoriel.

II.3.2. SOMME TENSORIELLE

Maintenant, A est la somme tensorielle L : $A = \bigoplus_{g_{i=1}}^{N} Q^{(i)}$.

Soit $x \in \hat{S}$ et $y \in \hat{S}$ deux états globaux. Avec la définition de la somme tensorielle généralisée (cf chapitre 2, section I.2 page 27), on a

$$a_{xy}(x) = \sum_{i=1}^{N} q^{(i)}_{x^{(i)}y^{(i)}}(x) \Delta_{(x^{(i)},y^{(i)})}$$

où[1] $\Delta_{(x^{(i)},y^{(i)})} = \prod_{j=1..N, j \neq i} \delta_{x^{(j)}y^{(j)}}$.

[1] $\delta_{uv} = 1$ si $u = v$, et 0 sinon

Donc

$$\tilde{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \sum_{\sigma\in\mathcal{P}} a_{x\sigma(y)}(x) = \sum_{\sigma\in\mathcal{P}}\sum_{i=1}^{N} q^{(i)}_{x^{(i)}\sigma(y^{(i)})}(x)\Delta_{(x^{(i)},\sigma(y^{(i)}))}$$

On peut remarquer que les seules permutations $\sigma = (\sigma_1 \ ... \ \sigma_K)$ donnant un résultat non nul pour $\Delta_{(x^{(i)},\sigma(y^{(i)}))}$ sont telles que seulement l'une des permutations σ_h, $h = 1..K$ n'est pas la permutation identité (notée id).

Soit $\bar{\sigma}_h = (id \ ... \ \sigma_h \ ...id)$ *une telle permutation.* Alors on a :

$$\tilde{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \sum_{h=1}^{K}\left(\sum_{\sigma_h\in\mathcal{P}_h}\sum_{i=1}^{N} q^{(i)}_{x^{(i)}\bar{\sigma}_h(y^{(i)})}(x)\Delta_{(x^{(i)},\bar{\sigma}_h(y^{(i)}))}\right)$$

$\Delta_{(x^{(i)},\bar{\sigma}_h(y^{(i)}))}$ peut être non nul uniquement pour $i \in EI_h$, donc finalement

$$\tilde{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \sum_{h=1}^{K}\left(\sum_{\sigma_h\in\mathcal{P}_h}\sum_{i\in EI_h} q^{(i)}_{x^{(i)}\bar{\sigma}_h(y^{(i)})}(x)\Delta_{(x^{(i)},\bar{\sigma}_h(y^{(i)}))}\right) \tag{3.5}$$

On définit alors la matrice agrégée $\hat{A}$ (définition 23) par une somme tensorielle de K matrices :

$$\hat{A} = \bigoplus_{h\in[1..K]}^{g} \hat{A}^h$$

et la matrice $\hat{A}^h$, $h \in [1..K]$, est définie par

$$\hat{a}^h_{\tilde{x}_h\tilde{y}_h}(\tilde{x}) = \sum_{\sigma_h\in\mathcal{P}_h}\sum_{i\in EI_h} q^{(i)}_{x^{(i)}\sigma_h(y^{(i)})}(\tilde{x})\Delta^h_{(x^{(i)},\sigma_h(y_{(i)}))}$$

où $\Delta^h_{(x^{(i)},y^{(i)})} = \prod_{j\in EI_h, j\neq i}\delta_{x^{(j)}y^{(j)}}$.

Or $\hat{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \sum_{h=1}^{K}\hat{a}^h_{\tilde{x}_h\tilde{y}_h}$, d'où

$$\hat{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \sum_{h=1}^{K}\left(\sum_{\sigma_h\in\mathcal{P}_h}\sum_{i\in EI_h} q^{(i)}_{x^{(i)}\sigma_h(y^{(i)})}(\tilde{x})\Delta^h_{(x^{(i)},\sigma_h(y_{(i)}))}\right)\prod_{k\in[1..K],k\neq h}\delta_{x_k y_k} \tag{3.6}$$

Dans la formule 3.6, le produit $\prod_{k\in[1..K],k\neq h}\delta_{x_k y_k}$ multiplié par les $\Delta^h_{(x^{(i)},\sigma_h(y_{(i)}))}$ est équivalent aux $\Delta_{(x^{(i)},\bar{\sigma}_h(y^{(i)}))}$ définis pour la formule 3.5.

Les deux formules sont donc identiques, ce qui prouve la formule 3.2 ($\tilde{a}_{\tilde{x}\tilde{y}}(\tilde{x}) = \hat{a}_{\tilde{x}\tilde{y}}(\tilde{x})$) dans le cas où A est une somme tensorielle.

II.3.3. EXPRESSION TENSORIELLE DE $\tilde{Q}$

On a donc démontré que pour chaque matrice A (définition 23), $\tilde{A} = \hat{A}$. On a donc $\tilde{Q} = \hat{L} + \sum_{e \in \mathcal{ES}} (\hat{P}_e + \hat{N}_e)$, et grâce à la définition des différentes matrices agrégées $\hat{A}$,

$$\tilde{Q} = \bigoplus_{h \in [1..K]}{}_g \, \hat{L}^h + \sum_{e \in \mathcal{ES}} \left(\bigotimes_{h \in [1..K]}{}_g \, \hat{P}_e{}^h + \bigotimes_{h \in [1..K]}{}_g \, \hat{N}_e{}^h \right)$$

Toutes les matrices $\hat{A}^h$ ont été définies précédemment.

On obtient ainsi une représentation tensorielle compacte de la chaîne de Markov agrégée. On a procédé comme si chaque groupe avait été agrégé de façon indépendante.

III. LES BÉNÉFICES DE L'AGRÉGATION

Nous avons prouvé que l'on peut agréger fortement un SAN avec plusieurs répliquas, mais il reste à montrer les bénéfices que l'on peut tirer d'une telle agrégation pour justifier l'intérêt de cette agrégation.

On présentera tout d'abord quelques résultats théoriques avant de montrer les bénéfices de l'agrégation sur des exemples pratiques.

III.1. RÉSULTATS THÉORIQUES

Si l'on considère un **SAN composé d'un seul répliqua** avec N automates, soit M le nombre d'états par automate (c'est le même pour chaque automate vu que ce sont des automates répliqués). Le nombre total d'états du SAN est alors $|\hat{S}| = M^N$. On cherche à calculer le nombre d'états de la chaîne de Markov agrégée, $|\hat{S}_{ag}|$. Noter que l'on ne prend pas en compte les états non accessibles, et certains états de la chaîne de Markov agrégée peuvent être supprimés (comme dans l'exemple RS1-1 présenté en section II.1). Ainsi, on donne une borne supérieure du nombre d'états de la chaîne de Markov agrégée.

Pour un état global donné du SAN initial, on note par na_m ($m \in [1..M]$) le nombre d'automates qui sont dans l'état local m (on numérote les états locaux de 1 à M). Le nombre d'états de la chaîne de Markov agrégée est égal au nombre de classes d'équivalence, et une classe d'équivalence est définie par un ensemble de na_m, avec $m \in [1..M]$. En effet, deux états globaux sont équivalents s'il y a le même nombre d'automates dans chaque état local $m \in [1..M]$. Le nombre d'états de la chaîne de Markov agrégée $|\hat{S}_{ag}|$ est donc borné par le

nombre de solutions entières de l'équation

$$\sum_{m=1}^{M} na_m = N$$

Cette condition doit être vérifiée vu qu'il y a N automates. La solution [52] peut être interprétée comme la disposition de $M-1$ pièces (numérotées de 1 à $M-1$) dans un ensemble de places adjacentes. Alors le nombre de places vides entre la pièce $(m-1)$ et la pièce m est na_m, pour $m = 2..M-1$. na_1 est le nombre de places vides avant la première pièce, et na_M est le nombre de places vides après la dernière pièce (cf Figure 3.3).

Il y a $M-1$ places pour les pièces, et $\sum_{m=1}^{M} na_m = N$ places vides, ce qui fait un total de $N+M-1$ places. Le nombre d'états de la chaîne de Markov agrégée est donc

$$|\hat{S}_{ag}| = \binom{N+M-1}{M-1} = \frac{(N+M-1)!}{N!(M-1)!}$$

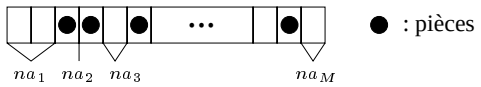


FIG. 3.3 – Exemple avec $na_1 = 2$, $na_2 = 0$, $na_3 = 1$, ..., $na_M = 1$.

Pour un **SAN avec plusieurs répliquas**, on a pour chaque groupe $g \in [1..K]$ $\frac{(R_g+M_g-1)!}{R_g!(M_g-1)!}$ états agrégés (M_g est le nombre d'états des automates du groupe g, et R_g le nombre d'automates identiques dans ce groupe). Cela fait donc un nombre total d'états :

$$|\hat{S}_{ag}| = \prod_{g=1}^{K} \frac{(R_g+M_g-1)!}{R_g!(M_g-1)!}$$

Par exemple, si l'on a 2 groupes constitués de R automates identiques, chacun ayant 2 états, et K autres groupes réduits à un automate ayant M états, on a initialement $2^R \times 2^R \times M^K$, et seulement $(R+1) \times (R+1) \times M^K$ états après agrégation.

III.2. EXEMPLES PRATIQUES

Nous donnons maintenant quelques valeurs numériques pour illustrer la réduction du nombre d'états induite par l'agrégation. Ensuite, nous présentons des résultats sur les exemples de modélisation exposés dans le chapitre 2.

III.2.1. QUELQUES CHIFFRES

On remarque tout d'abord que l'agrégation est d'autant plus intéressante qu'il y a beaucoup d'automates. Ainsi, si on agrège un SAN avec $N = 2$ et $M = 5$, on a initialement $5^2 = 25$ états, et on obtient 15 états après agrégation.

Pour de grandes valeurs de N, les résultats sont plus intéressants. Par exemple, lorsque $N = 100$ et $M = 5$, on obtient environ 4.6 million d'états après agrégation, au lieu des $5^{100} \approx 10^{70}$ états initiaux.

Si on a N automates à 2 états chacun, on a initialement 2^N états, et seulement $N + 1$ après agrégation.

L'agrégation est très utile pour les SANs à grand espace d'état, et on observe que dans ces cas, la réduction de l'espace d'état est très importante.

Les courbes de la figure 3.4 illustrent la réduction de l'espace d'état. C'est le pourcentage de la taille de l'espace d'état agrégé comparé à la taille de l'espace d'état initial, fonction de N, et pour différentes valeurs de M (on se place dans le cadre de SANs composés d'un seul répliqua). L'augmentation de la taille de l'espace d'état que l'on observe pour $N < 1$ ne doit pas être prise en compte vu que seules les valeurs entières de N sont significatives.

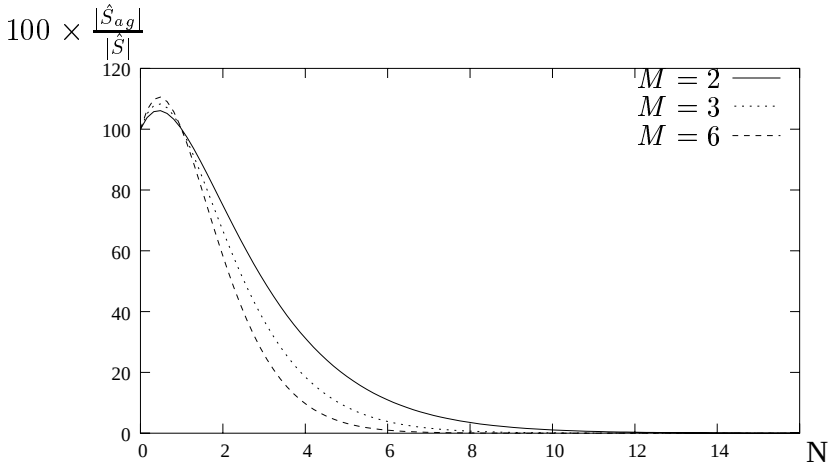


FIG. 3.4 – Réduction de l'espace d'état

Lorsque $N > 10$, la taille de l'espace d'état agrégé est négligeable en comparaison de la taille de l'espace d'état initial, et la réduction de l'espace d'état est d'autant plus importante que M est grand.

III.2.2. EXEMPLES DE MODÉLISATION

Nous reprenons ici les exemples de modélisation présentés dans le chapitre 2 (section IV page 53). On s'est limité aux exemples du partage de ressources, qui présentent des répliquas. Les exemples RS1-1 et RS1-2 ont été respectivement présentés dans les sections I.1.3 et I.2.3 de ce chapitre, pour illustrer la notion de SAN avec réplication.

Le tiret (-) dans le tableau signifie que l'on n'a pas pu calculer la valeur correspondante, du fait de sa complexité.

Modèle	SAN initial		chaîne de Markov agrégée	
	$\lvert\hat{S}\rvert$	$\lvert S\rvert$	$\lvert\hat{S}_{agg}\rvert$	$\lvert S_{agg}\rvert$
RS1-1 N_p, N_r	2^{N_p}	$\sum_{i=0}^{N_r}\binom{N_p}{i}$	N_p+1	N_r+1
RS1-1 $N_p = 20$, $N_r = 10$	$1,048,576$	$616,666$	21	11
RS1-1 $N_p = 10,000$, $N_r = 8,000$	$2^{10,000}$	-	$10,001$	$8,001$
RS1-2 $N_p = 20$, $N_r = 10$ $R_1 = 5$, $R_2 = 15$	$1,048,576$	$616,666$	96	51
RS3 N_p, N_r	$2 * 3^{N_p}$	$1+\sum_{i=0}^{N_r}\binom{N_p}{i}$	$(N_p+2)(N_p+1)$	N_r+2
RS3 $N_p = 20$, $N_r = 10$	$\approx 7 * 10^9$	$616,667$	462	12

Ces résultats montrent que l'agrégation produit toujours une réduction de l'espace d'état importante. Ceci peut ainsi permettre de ramener à une échelle raisonnable (de l'ordre du million d'états) des modèles à l'origine gigantesques, comme par exemple le modèle RS1-1 avec $N_p = 10,000$, ou le modèle RS3 avec $N_p = 20$.

IV. CONCLUSION

Nous avons défini dans ce chapitre les réseaux d'automates stochastiques avec réplication, et proposé une technique pour agréger de tels SANs. Pour cela, nous avons démontré un théorème disant comment une agrégation forte pouvait être appliquée au SAN. Nous avons également montré comment exprimer la matrice de la chaîne de Markov agrégée sous forme tensorielle. Des résultats théoriques et quelques valeurs numériques ont été présentés pour montrer que ces techniques d'agrégation sont très efficaces pour réduire la taille de l'espace d'états.

Une des limitations de ce travail est que les groupes d'automates répliqués doivent être définis lors de la spécification haut niveau. C'est cependant souvent un faux problème vu que les répliquas sont souvent connus lorsque l'on modélise un système particulier. Une détection automatique des répliquas ne représente pas de grandes difficultés d'implémentation, et un tel algorithme reste à écrire.

Après ces deux premiers chapitres qui ont présenté des techniques de modélisation à l'aide de réseaux d'automates stochastiques, la suite de cette thèse se concentre sur les méthodes pour calculer des indices de performance à partir de modèles SANs à temps continu, et notamment à la multiplication d'un vecteur par le descripteur d'un SAN.

Nous présenterons alors dans le chapitre 5 des résultats numériques (temps d'exécution pour obtenir des indices de performances, ...) obtenus avec les techniques d'agrégation de SANs avec répliquas.

CHAPITRE

Multiplication Vecteur-Descripteur

4

Pour obtenir des indices de performances sur un système Markovien, on est amené à calculer le vecteur des probabilités stationnaires. Les systèmes ciblés par les réseaux d'automates stochastiques sont les systèmes à grand espace d'états, et ce sont alors les méthodes itératives qui sont le plus adéquates (cf chapitre 1, page 11).

L'opération de base des méthodes itératives est la multiplication du vecteur par le générateur de la chaîne de Markov, et ces méthodes ont souvent besoin de calculer ce produit un grand nombre de fois. Leur efficacité dépend donc directement de la façon dont les données sont stockées.

Ainsi, durant la génération de la chaîne de Markov, une attention toute particulière doit être portée au stockage de l'espace d'états accessibles S, des vecteurs de probabilité π, et du générateur Q. En effet, même si le temps de résolution d'une méthode numérique peut être relativement élevé, ce sont surtout les considérations de place mémoire qui déterminent si cette méthode peut être employée ou non.

Au niveau du stockage de l'**espace d'états accessibles** S, il faut une structure permettant d'avoir un algorithme de recherche efficace. Il est parfois possible de définir une fonction d'accessibilité pour le modèle, ce qui présente le gros avantage d'offrir un accès aux états accessibles en temps constant. Cette approche est notamment utilisée dans le logiciel PEPS (cf chapitre 5, section I). D'autres techniques, basées sur une approche à plusieurs niveaux, comme par exemple les diagrammes de décision [32] peuvent également s'avérer être très efficaces. Par la suite, nous supposons que S est stocké de façon efficace, et nous ne nous préoccupons plus de savoir comment il est stocké.

De nombreuses techniques permettent de représenter le **générateur** Q et les **vecteurs de probabilité**.

Stockage explicite, matrice en format creux

Une première approche, indépendante du formalisme de haut niveau employé, consiste à stocker la matrice générateur de façon explicite en ne gardant que les éléments non nuls et leur position dans la matrice, par exemple en format Harwell-Boeing. Les vecteurs de probabilité sont alors de la taille de l'espace d'états accessibles S. Des algorithmes efficaces sont disponibles pour calculer un produit vecteur-matrice quand la matrice est stockée de cette manière [84, 90].

Génération à la volée, disque secondaire

Sanders et ses collaborateurs [36] proposent de générer les éléments de Q lorsqu'ils sont requis, génération qui a lieu à la volée depuis un formalisme de haut niveau. Cependant, cette approche peut être très coûteuse au niveau des temps à l'exécution, surtout lorsque

le modèle comprend des transitions immédiates. Une autre solution consiste à stocker Q sur un disque secondaire rapide, et à aller y chercher les éléments à la demande [37]. Ceci nécessite bien évidemment un disque suffisamment rapide, et qui soit de capacité suffisante.

Approche de Kronecker

Les réseaux d'automates stochastiques, introduits précédemment, permettent de diminuer les besoins en mémoire. Nous avons en effet vu dans le chapitre 2 (section III.2.2 page 40) que nous disposons d'une manière compacte de stockage, basée sur une formule tensorielle. On parle alors de *descripteur*.

Une approche similaire basée sur le formalisme des réseaux de Pétri stochastiques permet d'obtenir une formule tensorielle équivalente, comme l'a montré Donatelli dans [38, 39].

Cependant, cette approche basée sur l'utilisation de composants indépendants connectés par des synchronisations et des fonctions génère souvent une représentation avec beaucoup d'états non accessibles. $\hat{S}$ est l'espace d'états produit induit par cette approche, et on a souvent $|S| \ll |\hat{S}|$.

Dans le cadre de cette approche Kronecker, de nombreux algorithmes ont été proposés. Le premier, et sans doute le plus connu, est l'algorithme du shuffle [41, 42, 81, 7], qui calcule le produit vecteur-descripteur sans jamais exprimer la matrice de façon explicite. Cependant, il a été montré précédemment [41, 81] que cet algorithme requiert l'utilisation de vecteurs de probabilité $\hat{\pi}$ de la taille de l'espace d'états produit $\hat{S}$. Ces vecteurs sont appelés *vecteurs étendus*. Ainsi, nous dénotons l'algorithme du shuffle classique par **E-Sh** (E pour *étendu*).

Des approches alternatives ont été proposées dans [67, 66, 20, 7, 8]. L'idée consiste à calculer dans un premier temps l'espace d'états accessibles S, puis à résoudre le modèle en utilisant des vecteurs d'itération π ne contenant des entrées que pour les états accessibles (vecteurs de la taille de S). Par opposition aux vecteurs étendus, ces vecteurs sont appelés *vecteurs réduits*.

Diagrammes de décision

Une autre approche consiste à utiliser des diagrammes de décision pour représenter le générateur de la chaîne de Markov Q. Cette représentation arborescente du générateur ne garde en mémoire que les éléments distincts non nuls, qui se trouvent sur les feuilles de l'arbre. Lors de la construction, les branches de l'arbre doivent être coupées pour éviter qu'il devienne trop large. Ainsi, les diagrammes de décision binaires multi-terminaux (MTBDDs) et les graphes de décision probabilistes (PDGs) peuvent être utilisés à la fois en vérification de modèles ([63, 9]) et en évaluation de performance ([58]).

Cette approche est prometteuse, mais son efficacité en mémoire dépend directement du nombre d'éléments distincts de Q. De plus, l'accès aux éléments non nuls nécessite de suivre un chemin depuis la racine de l'arbre jusqu'à la bonne branche, et cette opération doit être effectuée chaque fois que l'on a besoin d'un élément.

Nous ne connaissons pas à ce jour d'étude comparative sur l'efficacité de ces algorithmes pour la multiplication vecteur-générateur.

Diagrammes de matrices

Dans un souci d'amélioration des temps d'exécution, une autre sorte de représentation est présentée dans [30, 31, 75]. Cette représentation du générateur Q sous la forme d'un diagramme de matrices permet d'avoir un accès rapide à chaque élément, et le temps de calcul est souvent satisfaisant, même pour les modèles à grand espace d'états. De plus, les besoins en mémoire pour stocker Q sont à peine supérieur à ceux requis pour stocker un descripteur sous forme tensorielle, et restent négligeable par rapport à la place mémoire occupée par un vecteur de probabilité.

Cependant, l'utilisation de techniques liées à un cache [31] rendent une estimation de la complexité théorique des algorithmes difficile, alors que de bons résultats théoriques peuvent être obtenus par l'approche Kronecker [20, 21].

Motivations

Lorsque $|S| \approx |\hat{S}|$, le gain mémoire obtenu par le formalisme tensoriel peut être énorme en comparaison de l'approche d'un stockage explicite du générateur sous forme de matrice creuse. Par exemple, si un modèle est constitué de N composants de taille n_i ($i = 1..N$), et si le générateur est plein (pas d'éléments nuls), la place mémoire nécessaire pour le stocker explicitement est de l'ordre de $(\prod_{i=1}^{N} n_i)^2$. L'utilisation d'un formalisme tensorielle réduit ce coût à $\sum_{i=1}^{N} n_i^2$. Dans ce cas, l'algorithme du shuffle E-Sh est très efficace, comme cela a été montré dans [41, 42, 80, 7, 8].

Cependant, lorsque le nombre d'états non accessibles est élevé ($|S| \ll |\hat{S}|$), E-Sh n'est pas efficace à cause de l'utilisation de vecteurs étendus. Le vecteur de probabilité peut alors avoir un grand nombre d'éléments nuls, car seuls les éléments correspondant à des états accessibles ont une probabilité non nulle. De plus, les calculs doivent être effectués pour tous les éléments du vecteur, y compris pour les éléments correspondant à des états non accessibles. Ainsi, le gain obtenu par l'utilisation du formalisme tensoriel peut être perdu car de nombreux calculs inutiles sont effectués, et de l'espace mémoire est utilisé pour des états dont la probabilité est toujours nulle. Dans ce cas, l'approche basée sur un stockage explicite de tous les éléments non nuls du générateur est meilleure car elle n'effectue pas de calculs inutiles. Cependant, ses besoins en mémoire élevés empêchent son application sur les modèles très grands.

A cause des grands besoins en mémoire requis par un stockage explicite du générateur, il nous semble intéressant de rechercher une solution qui prenne en compte les états non accessibles et qui, en même temps, tire profit du formalisme tensoriel. Ainsi, nous aimerions exploiter le formalisme tensoriel même en présence d'un grand nombre d'états non accessibles. En fait, l'algorithme E-Sh est très efficace lorsque $|S| \approx |\hat{S}|$, et on va montrer comment le rendre efficace dans le cas où $|S| \ll |\hat{S}|$. Il nous a semblé intéressant d'approfondir les optimisations sur cet algorithme qui exploite la structure tensorielle et pour lequel on possède certains résultats théoriques. Le formalisme tensoriel connaît par ailleurs un regain d'intérêt dans différents domaines d'application, comme par exemple en traitement du signal [53], dans les équations différentielles partielles, ...

L'utilisation de *vecteurs réduits* permet de réduire les besoins en mémoire, et des calculs inutiles peuvent alors être évités. Ceci permet alors d'obtenir des gains en mémoire très important lors de l'utilisation de méthodes itératives comme Arnoldi ou GMRES, qui peuvent

nécessiter un grand nombre de vecteurs de probabilité. Une modification de l'algorithme du shuffle E-Sh permet l'utilisation de tels vecteurs. Ainsi, l'algorithme Act-Sh-JCB de [20] transforme le vecteur réduit π en un vecteur étendu $\hat{\pi}$ de la taille de $\hat{S}$ avant d'appeler E-Sh, mais la plupart des vecteurs utilisés sont réduits.

Plan du chapitre

Nous commençons par décrire le classique algorithme du shuffle E-Sh, qui permet de multiplier un vecteur de probabilité par le descripteur d'un réseau d'automates stochastiques. Cet algorithme est efficace lorsqu'il y a peu d'états non accessibles dans le modèle.

Des modifications ont été apportées à cet algorithme pour réduire les besoins en mémoire lorsque $|S| \ll |\hat{S}|$ [20, 7]. Nous proposons dans la section II un algorithme [7] qui permet d'obtenir un gain de temps à l'exécution, en exploitant le fait que les probabilités des éléments correspondant à des états non accessibles sont toujours nulles dans les vecteurs de probabilité. Cependant, pour obtenir de bonnes performances au niveau du temps d'exécution, des vecteurs intermédiaires de la taille de $\hat{S}$ doivent également être utilisés. Nous appelons cet algorithme le shuffle **partiellement réduit**, et on s'y réfère par **PR-Sh**. Cependant, les gains en mémoire se trouvent relativement limités car on utilise toujours des structures de données de la taille de $\hat{S}$.

Conserver la bonne complexité de E-Sh tout en éliminant l'utilisation de toutes structures de la taille de $\hat{S}$ n'a à notre connaissance jamais été obtenu. Nous avons donc développé un tel algorithme [8] pour réduire les besoins en mémoire, tout en limitant les coûts supplémentaires à l'exécution. L'algorithme du shuffle **totalement réduit FR-Sh** (*fully reduced*) est décrit dans la section III. Dans cet algorithme, toutes les structures de données intermédiaires sont stockées dans un format réduit. Il permet ainsi de traiter des modèles de plus en plus complexes.

Nous exposons dans la section IV des variantes de ces algorithmes, basées sur un réordonnancement des automates, qui permettent d'optimiser le nombre de calculs.

I. L'ALGORITHME DU SHUFFLE "CLASSIQUE" : E-Sh

Le descripteur d'un SAN peut s'exprimer sous forme tensorielle (cf chapitre 2, section III.2.2 page 40) :

$$Q = D + \bigoplus_{i=1}^{N}{}_g\, Q_l^{(i)} + \sum_{e=1}^{E} \bigotimes_{i=1}^{N}{}_g\, Q_e^{(i)} = \sum_{j=1}^{(N+2E)} \left[\bigotimes_{i=1}^{N}{}_g\, Q_j^{(i)} \right]$$

Dans la première représentation (analogue à la formule 2.7), la somme tensorielle correspond à l'analyse des événements locaux, c'est la partie locale du descripteur ; le produit tensoriel correspond à l'analyse des événements synchronisant, c'est la partie synchronisante du descripteur ; enfin D représente le produit tensoriel des matrices de synchronisation négatives, c'est la diagonale du descripteur.

La deuxième représentation (formule 2.8) montre que l'opération de base qui nous intéresse lors du calcul de la solution stationnaire d'une chaîne de Markov en utilisant des méthodes de résolution itératives est le produit d'un vecteur $\hat{\pi}$ et d'un terme produit tensoriel généralisé[1] (cf section I.1 page 24) :

$$\hat{\pi} \quad \bigotimes_{i=1}^{N}{}_g\, Q^{(i)} \tag{4.1}$$

Le produit tensoriel est composé d'une suite de N matrices notées $Q^{(i)}$ avec $i \in [1..N]$, chacune associée à un automate $\mathcal{A}^{(i)}$.

Nous commençons par introduire quelques définitions sur les suites finies de matrices[2] :

✍ **Soit**

n_i	la dimension de la i-ème matrice d'une suite ;
$nleft_i$	le produit des dimensions de toutes les matrices à gauche de la i-ème matrice d'une suite, *i.e.*, $\prod_{k=1}^{i-1} n_k$ (cas particulier : $nleft_1 = 1$) ;
$nright_i$	le produit des dimensions de toutes les matrices à droite de la i-ème matrice d'une suite, *i.e.*, $\prod_{k=i+1}^{N} n_k$ (cas particulier : $nright_N = 1$) ;
$njump_i$	le produit des dimensions de toutes les matrices à droite de la i-ème matrice d'une suite, y compris la matrice i, *i.e.*, $\prod_{k=i}^{N} n_k$ ($njump_i = nright_i \times n_i$) ;

[1] Les indices j ont été omis pour les matrices $Q_j^{(i)}$ afin de simplifier la notation.

[2] Par la suite, les *suites finies de matrices* seront appelées seulement *suites de matrices*. Car, seules les suites finies seront abordées.

Nous présentons tout d'abord l'algorithme de la multiplication d'un vecteur par un produit tensoriel classique. Cet algorithme est utilisé pour les descripteurs qui n'ont pas d'éléments fonctionnels. Ensuite, nous verrons les contraintes à observer lors de l'occurrence des éléments fonctionnels dans le terme produit tensoriel (produits tensoriels généralisés).

I.1. CAS SANS ÉLÉMENTS FONCTIONNELS

Le cas le plus simple de la multiplication d'un vecteur par un produit tensoriel est celui où les matrices ne contiennent pas d'éléments fonctionnels. Dans ce cas, c'est un produit tensoriel classique, et on doit calculer :

$$\hat{\pi} \bigotimes_{i=1}^{N} Q^{(i)}$$

Selon la propriété de décomposition des produits tensoriels [41], tout produit tensoriel de N matrices est équivalent au produit de N facteurs normaux. En utilisant cette propriété pour le terme $\bigotimes_{i=1}^{N} Q^{(i)}$:

$$\begin{aligned} Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-1)} \otimes Q^{(N)} = \\ I_{nleft_1} \otimes Q^{(1)} \otimes I_{nright_1} \\ \times I_{nleft_2} \otimes Q^{(2)} \otimes I_{nright_2} \\ \times \cdots \\ \times I_{nleft_N} \otimes Q^{(N)} \otimes I_{nright_N} \end{aligned}$$

Pour calculer la multiplication d'un vecteur $\hat{\pi}$ par le terme $\bigotimes_{i=1}^{N} Q^{(i)}$ il est donc suffisant de savoir multiplier un vecteur par un facteur normal i ($i \in [1..N]$) :

$$\hat{\pi} \times I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$$

De plus, la propriété de commutativité entre facteurs normaux permet la multiplication des facteurs normaux dans un ordre quelconque.

I.1.1. MULTIPLICATION PAR UN FACTEUR NORMAL

On considère la multiplication par le facteur normal i, où $i \in [1..N]$. La matrice $I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$ est une matrice bloc diagonale dont les blocs sont constitués de la matrice $Q^{(i)} \otimes I_{nright_i}$. Les différents blocs peuvent être traités de façon indépendante, ce qui suggère la possibilité d'introduire du parallélisme dans l'algorithme, [92, 94, 93].

Il y a $nleft_i$ blocs de matrice, et chacun d'entre eux doit être multiplié par une portion différente de vecteur. Le vecteur est ainsi divisé en $nleft_i$ portions, chacune de taille $nright_i \times n_i$. Nous appelons ces portions de vecteur des l-portions (pour $left$), et nous les notons $p_0, \ldots, p_{nleft_i - 1}$. La figure 4.1 illustre cette découpe.

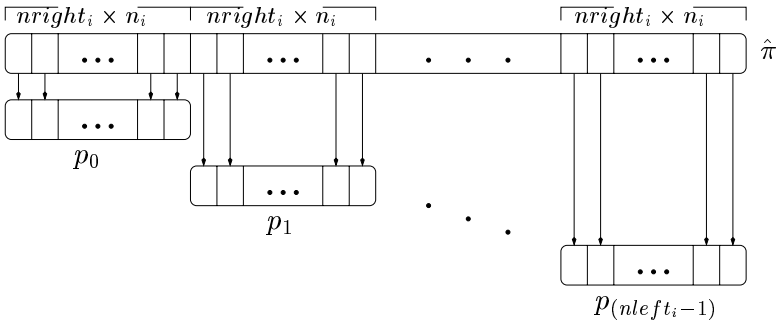


FIG. 4.1 – Découpe du vecteur $\hat{\pi}$ en l-portions

Ainsi, l'algorithme E-Sh consiste en une boucle sur les $nleft_i$ l-portions, et à chaque itération il calcule la l-portion résultat :

$$r_l = p_l \ \times \ Q^{(i)} \otimes I_{nright_i}, \ l = 0..nleft_i - 1$$

I.1.2. DÉTAIL D'UNE PORTION : CALCUL DE r_l

Observons le format de la matrice $Q^{(i)} \otimes I_{nright_i}$:

$$Q^{(i)} \otimes I_{nright_i} = \begin{pmatrix} q^{(i)}_{1,1} I_{nright_i} & q^{(i)}_{1,2} I_{nright_i} & \cdots & q^{(i)}_{1,n_i} I_{nright_i} \\ q^{(i)}_{2,1} I_{nright_i} & q^{(i)}_{2,2} I_{nright_i} & \cdots & q^{(i)}_{2,n_i} I_{nright_i} \\ \vdots & \vdots & \ddots & \vdots \\ q^{(i)}_{n_i,1} I_{nright_i} & q^{(i)}_{n_i,2} I_{nright_i} & \cdots & q^{(i)}_{n_i,n_i} I_{nright_i} \end{pmatrix}$$

Chaque matrice $q^{(i)}_{j,k} I_{nright_i}$ ($j, k \in [1..n_i]$) est une matrice diagonale de taille $nright_i$ dont tous les éléments de la diagonale sont égaux à $q^{(i)}_{j,k}$.

Le calcul d'un élément de r_l correspond à la multiplication de p_l par une colonne de la matrice $Q^{(i)} \otimes I_{nright_i}$. Or, chaque colonne de cette matrice est composée d'éléments d'une seule colonne de $Q^{(i)}$, les autres éléments sont zéro. La multiplication revient donc à extraire de façon répétée des éléments de p_l (distants de $nright_i$), à former un vecteur appelé z_{in} à partir de ces éléments, puis à multiplier z_{in} par une colonne de la matrice $Q^{(i)}$. Notez que z_{in} est composé d'éléments de p_l qui peuvent ne pas être consécutifs ; c'est une tranche de p_l. La tranche z_{in} correspond aux éléments de p_l qui doivent être multipliés par les éléments d'une colonne de $Q^{(i)}$. La colonne k ($k \in [1..n_i]$) de la matrice $Q^{(i)}$ est notée $q_{*,k}$.

La structure de la matrice $Q^{(i)} \otimes I_{nright_i}$ nous indique que nous devons considérer $nright_i$ tranches z_{in}. Nous numérotons donc les z_{in} de 0 à $nright_i - 1$. L'extraction d'un z_{in} revient à accéder au vecteur p_l et à choisir les éléments espacés de $nright_i$ positions dans le vecteur. On procède donc par sauts dans le vecteur. La figure 4.2 repré-

sente le processus d'extraction des z_{in}. Le r-ème z_{in} est représenté par $z_{in}\ n^\circ\ r$, pour $r = 0, \ldots, nright_i - 1$.

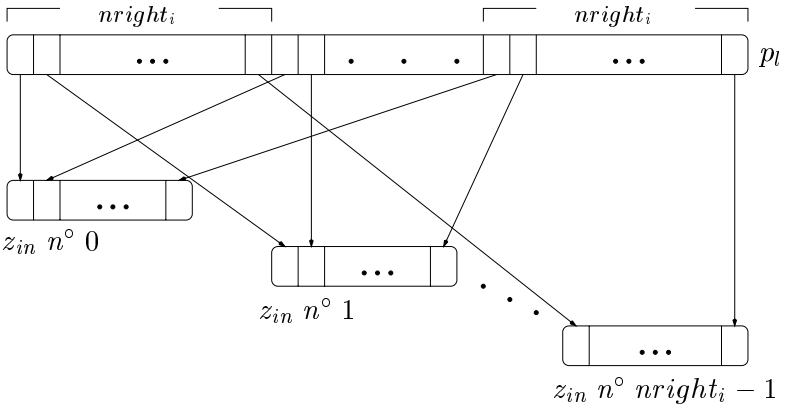


FIG. 4.2 – Découpe de p_l en z_{in}

Une fois un z_{in} obtenu, nous pouvons calculer un élément du résultat en multipliant le z_{in} par une colonne de $Q^{(i)}$ ($q_{*,k}$). La multiplication d'un z_{in} par la matrice $Q^{(i)}$ en entier donne ainsi plusieurs éléments du résultat. Ces éléments constituent un sous-vecteur de η ; ils forment une tranche de résultat, appelée z_{out}, et les positions des éléments de z_{out} dans r_l correspondent aux positions des éléments de z_{in} dans p_l. Nous numérotons les z_{out} de la même façon que les z_{in}. Ainsi, la multiplication du $z_{in}\ n^\circ\ r$ par $Q^{(i)}$ donne le $z_{out}\ n^\circ\ r$ ($r = 0..\ nright_i - 1$).

La figure 4.3 illustre la multiplication pour quelques éléments du vecteur résultat. Les éléments calculés constituent le $z_{out}\ n^\circ\ 0$. On effectue en effet la multiplication du $z_{in}\ n^\circ\ 0$ par toutes les colonnes de la matrice $Q^{(i)}$ pour obtenir successivement les différents éléments.

Pour traiter la portion (calculer $p_l \times Q^{(i)} \otimes I_{nright_i}$), on procède en effectuant une boucle pour extraire successivement les $nright_i$ z_{in}. Pour chaque z_{in}, nous effectuons la multiplication $z_{out} = z_{in} \times Q^{(i)}$, ce qui nous donne une tranche de vecteur résultat, puis nous stockons cette tranche z_{out} dans le vecteur résultat. Les positions dans le z_{out} étant les mêmes que celles du z_{in} correspondant, ce stockage se fait d'une façon analogue à l'extraction des z_{in} (cf figures 4.2 et 4.3).

I.1.3. ALGORITHME DE MULTIPLICATION

L'algorithme 4.1 résout la multiplication d'un vecteur $\hat{\pi}$ par un produit tensoriel $\otimes_{i=1}^{N} Q^{(i)}$. Nous notons cet algorithme **E-Sh sans fonction**, car nous ne traitons pas les taux fonctionnels. Dans cet algorithme les facteurs normaux sont traités du premier jusqu'au dernier. Pourtant, selon la propriété de commutativité de facteurs normaux, n'importe quel autre ordre aurait pu être employé.

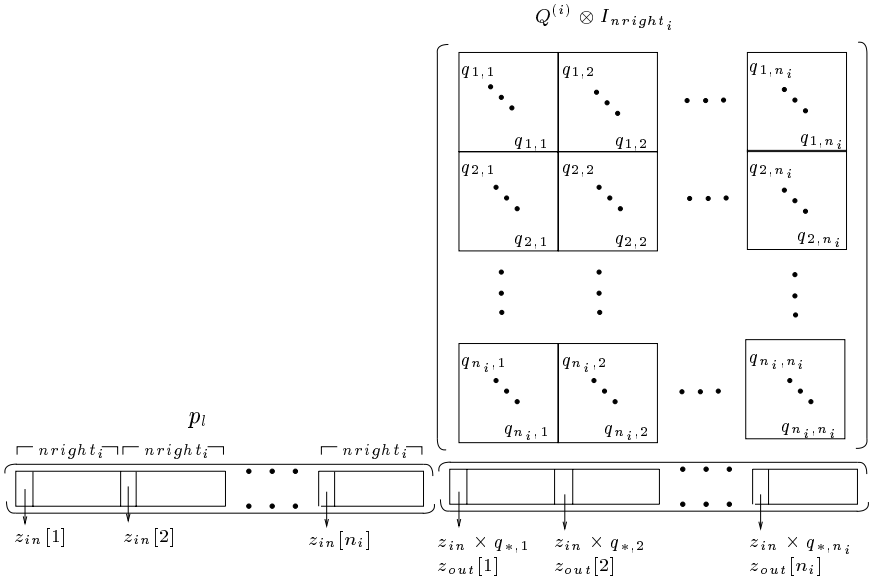


FIG. 4.3 – Multiplication $p_l \times Q^{(i)} \otimes I_{nright_i}$: obtention du z_{out} n° 0.

Algorithme 4.1

```
1   for i = 1, 2, ..., N          // boucle sur les facteurs normaux
2   do  base = 0 ;          // base : indice du début de la portion dans le vecteur
3       for l = 0, 1, ..., nleft_i − 1          // boucle sur les portions
4       do  for r = 0, 1, ..., nright_i − 1 // détail de la portion : boucle sur les z_in
5           do  index = base + r ;
            // index : indice de l'élément courant du z_in dans le vecteur
6               for k = 0, 1, ..., n_i − 1 // extraction du z_in n° r (sauts de nright_i)
7               do  z_in[k] = π̂[index] ;
8                   index = index + nright_i ;
9               end do
10              multiply    z_out = z_in × Q^(i) ;          // multiplication
11              index = base + r ;
                // index : indice de l'élément courant du z_out dans le vecteur
12              for k = 0, 1, ..., n_i − 1 // stockage du résultat z_out (sauts de nright_i)
13              do  π̂[index] = z_out[k] ;
14                  index = index + nright_i ;
15              end do
16          end do
17          base = base + ( nright_i × n_i ) ; // on passe à la portion suivante (saut dans π̂)
18      end do
19  end do
```

Algorithme 4.1: **E-Sh sans fonction**

I.1.4. COMPLEXITÉ

La complexité du produit d'un vecteur par un terme produit tensoriel classique est obtenue en observant le nombre de multiplications vecteur-matrice exécuté (ligne 10 de l'algorithme 4.1). À chaque boucle en i de l'algorithme, $nleft_i \times nright_i$ produits vecteur-matrice sont exécutés avec des matrices de taille n_i. En supposant les matrices $Q^{(i)}$ pleines, le nombre de multiplications pour chaque produit vecteur-matrice est égal à n_i^2. Il y a donc $nleft_i \times nright_i \times n_i^2 = \prod_{k=1}^{N} n_k \times n_i = |\hat{S}| \times n_i$ multiplications à chaque boucle en i ($|\hat{S}|$ est la taille de l'espace produit). La complexité de l'algorithme 4.1 est donc

$$|\hat{S}| \times \sum_{i=1}^{N} n_i \tag{4.2}$$

Ceci est à comparer avec une multiplication d'un vecteur par $Q = \bigotimes_{i=1}^{N} Q^{(i)}$ qui consiste d'abord à calculer Q, puis à multiplier le résultat par le vecteur. La complexité est ici de l'ordre de $(\prod_{i=1}^{N} n_i)^2 = |\hat{S}|^2$.

Si les matrices $Q^{(i)}$ sont stockées dans un format creux, le nombre de multiplications pour chaque produit vecteur-matrice est, en général, bien inférieur à $(n_i)^2$. Dans ce cas, si nz_i est le nombre d'éléments non nuls de la matrice $Q^{(i)}$, le nombre de multiplications à chaque boucle en i est $nleft_i \times nright_i \times nz_i = |\hat{S}| \times \frac{nz_i}{n_i}$. La complexité de l'algorithme 4.1 est donc :

$$|\hat{S}| \times \sum_{i=1}^{N} \frac{nz_i}{n_i} \tag{4.3}$$

Pour avoir une idée plus précise des coûts de calcul de cet algorithme, établissons une comparaison de cette complexité avec la multiplication d'un vecteur par une matrice unique stockée en format creux. La matrice équivalente à un terme produit tensoriel possédera $\prod_{i=1}^{N} nz_i$ éléments non nuls, donc la complexité sera de cet ordre. La comparaison entre deux formules aussi différentes est difficile. Fixons une limite de remplissage des matrices $Q^{(i)}$:

$$nz_i = n_i \times N^{\frac{1}{N-1}}$$

Pour ce nombre d'éléments non nuls les complexités de l'algorithme 4.1 (equation 4.3) et de la multiplication par une matrice creuse ($\prod_{i=1}^{N} nz_i$) sont égales. Pour des valeurs de nz_i inférieures à cette borne la multiplication par une matrice creuse sera plus performante et pour des valeurs supérieures l'avantage sera à l'algorithme proposé[3].

Ainsi, pour un modèle à 2 automates, s'il y a plus de $2n_i$ éléments non nuls par matrice (soit deux éléments non nuls par ligne de matrice), l'algorithme E-Sh sera plus performant qu'une multiplication par une matrice creuse. Pour 10 automates, la limite passe à $1.3n_i$.

La figure 4.4 présente la limite d'efficacité de E-Sh, en terme de nombre d'éléments non nuls par ligne de matrice. Ainsi, lorsqu'il y a plus d'un à deux éléments non nuls par ligne, c'est E-Sh qui est plus performant.

[3]Pour ces comparaisons, le coût de génération d'une matrice à partir d'une expression tensorielle n'est pas pris en compte.

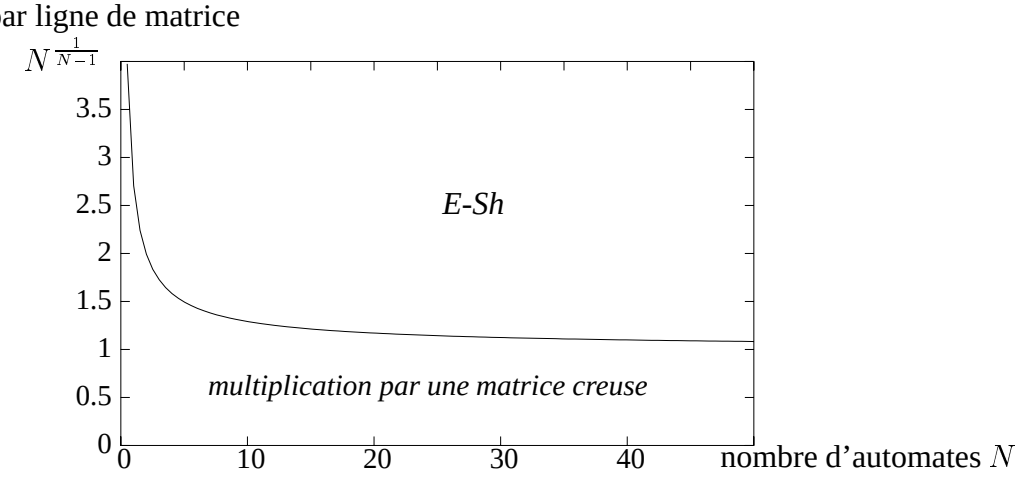


FIG. 4.4 – Limite d'efficacité de E-Sh, fonction du nombre d'automates N

Il faut rappeler que ces considérations sont applicables à la comparaison d'un seul terme produit tensoriel classique (sans éléments fonctionnels). Nous nous intéressons maintenant aux modifications à apporter pour traiter ces dépendances fonctionnelles.

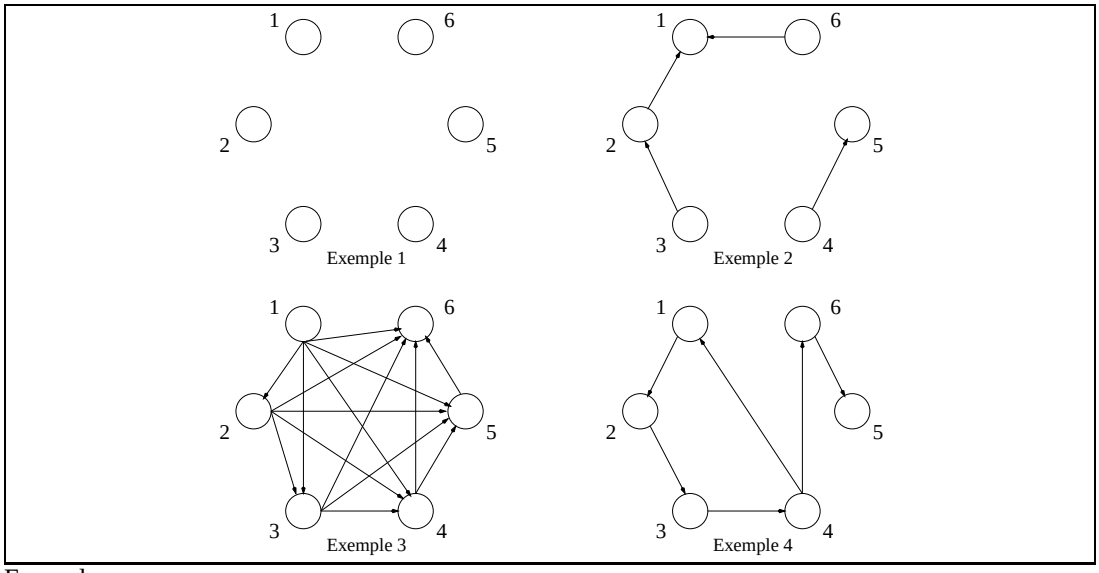
I.2. TRAITEMENT DES DÉPENDANCES FONCTIONNELLES

Lorsque les matrices $Q^{(i)}$ contiennent des éléments fonctionnels, le descripteur est constitué de produits tensoriels généralisés, comme dans l'équation 4.1.

D'après les propriétés de décomposition en facteurs normaux des produits tensoriels généralisés [41], il est toujours possible d'obtenir un ordre σ pour multiplier les facteurs normaux d'un terme $\otimes_{i=1}^{N} Q^{(i)}(\ldots)$ si et seulement s'il n'y a pas de cycles dans son *graphe de dépendances fonctionnelles*. Ce graphe orienté est défini pour les matrices d'un produit tensoriel généralisé. Les sommets du graphe correspondent à une matrice, et un arc d'un sommet i vers un sommet j représente la dépendance fonctionnelle $Q^{(i)}(\mathcal{A}^{(j)})$. La figure 4.5 montre quelques exemples de graphes de dépendance fonctionnelles.

L'existence de cycles empêche l'application directe des propriétés de décomposition en facteurs normaux. Le traitement nécessaire pour les éliminer est détaillé dans [41].

Dans cette section, nous présentons uniquement la multiplication des produits tensoriels généralisés sans cycle de dépendances fonctionnelles.



Exemples

1. $Q^{(1)} \underset{g}{\otimes} Q^{(2)} \underset{g}{\otimes} Q^{(3)} \underset{g}{\otimes} Q^{(4)} \underset{g}{\otimes} Q^{(5)} \underset{g}{\otimes} Q^{(6)}$
2. $Q^{(1)} \underset{g}{\otimes} Q^{(2)}(\mathcal{A}^{(1)}) \underset{g}{\otimes} Q^{(3)}(\mathcal{A}^{(2)}) \underset{g}{\otimes} Q^{(4)}(\mathcal{A}^{(5)}) \underset{g}{\otimes} Q^{(5)} \underset{g}{\otimes} Q^{(6)}(\mathcal{A}^{(1)})$
3. $\bigotimes_{g}^{6}{}_{i=1} Q^{(i)}(\mathcal{A}^{(i+1)}, \ldots, \mathcal{A}^{(6)})$
4. $Q^{(1)}(\mathcal{A}^{(2)}) \underset{g}{\otimes} Q^{(2)}(\mathcal{A}^{(3)}) \underset{g}{\otimes} Q^{(3)}(\mathcal{A}^{(4)}) \underset{g}{\otimes} Q^{(4)}(\mathcal{A}^{(1)}, \mathcal{A}^{(6)}) \underset{g}{\otimes} Q^{(5)} \underset{g}{\otimes} Q^{(6)}(\mathcal{A}^{(5)})$

FIG. 4.5 – Exemples de graphes de dépendances fonctionnelles

La multiplication d'un vecteur $\hat{\pi}$ par un produit tensoriel $\bigotimes_{g_{i=1}}^{N} Q^{(i)}(\mathcal{A}^{(i+1)}, \ldots, \mathcal{A}^{(N)})$ sans cycle est faite de façon similaire au cas sans éléments fonctionnels. Deux modifications doivent êtres faites à la multiplication implémentée par l'algorithme 4.1 :

– calculer un ordre σ dans lequel les facteurs normaux doivent être multipliés ;
– évaluer les éléments fonctionnels des matrices avant chacune de leur multiplication.

Nous détaillons maintenant ces deux aspects.

I.2.1. ÉTABLISSEMENT DE L'ORDRE DES FACTEURS NORMAUX

L'ordre peut être obtenu à partir du graphe de dépendances fonctionnelles [41, 42], en appliquant les propriétés de décomposition en facteurs normaux en algèbre tensorielle généralisée. Cet ordre est noté par une permutation σ sur l'intervalle $[1..N]$, appelé *ordre de décomposition*. Il représente l'ordre dans lequel le produit tensoriel sera décomposé en facteurs normaux.

✍ **Notations**

σ une permutation nommée σ sur l'intervalle $[1..N]$, qui établit un nouvel ordre pour une suite de N matrices ;

$\sigma(i)$ le rang de la matrice $Q^{(i)}$ dans l'ordre représenté par la permutation σ ;

σ_k l'indice de la matrice placée au rang k de l'ordre représenté par la permutation σ (si $\sigma_k = i$, $\sigma(i) = k$) ;

Le produit $\hat{\pi} \times \bigotimes_{g\, i=1}^{N} Q^{(i)}(\ldots)$ doit être traité par

$$\hat{\pi} \times \prod_{i=1}^{N} \left[I_{nleft_{\sigma_i}} \underset{g}{\otimes} Q^{(\sigma_i)}(\ldots) \underset{g}{\otimes} I_{nright_{\sigma_i}} \right]$$

La modification apportée à l'algorithme 4.1 pour prendre en compte ce changement d'ordre des facteurs normaux est indiquée dans l'algorithme 4.2. Notons que cette modification demande le calcul d'un ordre de traitement (permutation σ) qui n'est pas inclus dans l'algorithme de multiplication des facteurs normaux.

Algorithme 4.2

```
1  for i = σ_1, σ_2, ..., σ_N
2  do ···
          cf Algorithme 4.1
⋮
```

Algorithme 4.2: Changement de l'ordre des facteurs normaux

Pour certains produits tensoriels l'*ordre de décomposition* peut ne pas être unique. Par exemple, deux matrices dans la série peuvent être constantes. Dans ce cas, n'importe laquelle des matrices peut être traitée avant l'autre. Ceci est dû au fait que le produit de facteurs normaux de deux matrices constantes est commutatif. La règle générale est que deux (ou plusieurs) matrices n'ayant pas de dépendances fonctionnelles directes ou indirectes entre elles peuvent changer librement de position. Cette règle généralise l'absence d'un ordre précis pour la multiplication des facteurs normaux d'un produit tensoriel classique (cas sans fonction).

I.2.2. ÉVALUATION DES ÉLÉMENTS FONCTIONNELS

Après le calcul de l'*ordre de décomposition*, la deuxième préoccupation pour la multiplication des produits tensoriels généralisés est l'évaluation des éléments fonctionnels de chaque matrice $Q^{(i)}(\ldots)$ avant leur multiplication par une tranche z_{in} du vecteur $\hat{\pi}$ (correspondant à la ligne 10 de l'algorithme 4.1). Pour cette présentation, on supposera que l'ordre de décomposition σ est l'ordre $1..n$ pour simplifier les notations.

À chaque exécution de la multiplication d'une tranche z_n du vecteur $\hat{\pi}$ par la matrice $Q^{(i)}(\ldots)$, cette matrice doit être évaluée pour les états des automates arguments de $Q^{(i)}$.

Pour l'évaluation des éléments fonctionnels, il est donc nécessaire d'inclure le calcul des états locaux des automates arguments de $\mathcal{A}^{(i)}$. Comme ces arguments sont un sous ensemble à priori arbitraire du SAN, ceci amène à calculer l'état local de tout automate autre que $\mathcal{A}^{(i)}$. Ensuite il faut faire l'évaluation de $Q^{(i)}(\ldots)$ avec ces états locaux.

Examinons d'abord le calcul des états locaux des automates à gauche[4] de l'automate $\mathcal{A}^{(i)}$. L'indice l de l'algorithme 4.1 est le rang dans l'espace d'états $\prod_{j=1}^{i-1} S^{(j)}$, *i.e.*, des sous vecteurs d'états locaux du type $(x^{(1)}, x^{(2)}, \ldots, x^{(i-1)})$. Il est possible de regarder un sous vecteur d'états locaux comme un nombre en base variable, *i.e.*, un nombre pour lequel chacun des chiffres a sa propre base. L'état local de l'automate $\mathcal{A}^{(j)}$ $(x^{(j)})$ représentera un *nombre* qui peut varier dans l'intervalle $[0..n_j - 1]$. La suite de ces nombres permet d'obtenir le rang l dans l'espace d'états $\prod_{j=1}^{i-1} S^{(j)}$ d'un sous vecteur $(x^{(1)}, x^{(2)}, \ldots, x^{(i-1)})$ par la formule :

$$l = \sum_{j=1}^{i-1} \left(x^{(j)} \times \prod_{k=1}^{j-1} (n_k) \right)$$

De façon analogue, on peut obtenir le sous vecteur $(x^{(1)}, x^{(2)}, \ldots, x^{(i-1)})$ à partir d'un indice l donné, par une suite de divisions entières. Un algorithme qui implante ces divisions est très coûteux (autant de divisions entières que le nombre d'automates à gauche de $\mathcal{A}^{(i)}$). Une solution moins coûteuse peut être employée. La première remarque utile est que toutes les combinaisons d'états locaux vont être nécessaires. De plus, les combinaisons seront prises exactement dans l'ordre lexicographique (la valeur de l'indice l varie de 0 jusqu'à $nleft_i - 1$). La deuxième remarque intéressante est qu'il est peu coûteux de passer d'un sous vecteur d'états locaux au sous vecteur suivant selon un ordre lexicographique. Ceci peut être fait par un algorithme simple d'incrémentation de +1 d'un nombre en base variable.

Prenons par exemple, les états locaux de trois automates ($\mathcal{A}^{(1)}$, $\mathcal{A}^{(2)}$ et $\mathcal{A}^{(3)}$) avec tailles $n_1 = 2$, $n_2 = 4$ et $n_3 = 3$. Les sous vecteurs d'états locaux ordonnés selon l'ordre lexicographique sont :

rang	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	rang	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	rang	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	rang	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
(l)	(l_1)	(l_2)	(l_3)	(l)	(l_1)	(l_2)	(l_3)	(l)	(l_1)	(l_2)	(l_3)	(l)	(l_1)	(l_2)	(l_3)
0	0	0	0	6	0	2	0	12	1	0	0	18	1	2	0
1	0	0	1	7	0	2	1	13	1	0	1	19	1	2	1
2	0	0	2	8	0	2	2	14	1	0	2	20	1	2	2
3	0	1	0	9	0	3	0	15	1	1	0	21	1	3	0
4	0	1	1	10	0	3	1	16	1	1	1	22	1	3	1
5	0	1	2	11	0	3	2	17	1	1	2	23	1	3	2

Le changement de $\{0, 1, 1\}$ (de rang 4), vers le prochain sous vecteur dans l'ordre lexicographique (de rang 5) nécessite l'incrément de l'état du dernier état local (de 1 vers 2). Le changement de cette nouvelle combinaison ($\{0, 1, 2\}$ - rang 5) vers la prochaine (de rang

[4]Nous appelons *à gauche* d'une matrice d'indice i, toutes matrices d'un même terme qui ont des indices j inférieurs à i ($j \in [1..i-1]$).

6) ne correspond pas à l'incrément du dernier état local (il est déjà à sa valeur maximale $n_3 - 1 = 2$), donc l'avant dernier état local doit être incrémenté, ce qui donne $\{0, 2, 0\}$.

De façon générale, pour parcourir les sous vecteurs avec les états locaux des automates à gauche de $Q^{(i)}$ on initialise :

$$l_1 = 0 \quad l_2 = 0 \quad \cdots \quad l_{i-2} = 0 \quad l_{i-1} = 0$$

À chaque incrément de l, correspond une addition de +1 sur le nombre en base variable, soit un incrément de l'état local l_{i-1}, avec une propagation de retenue si nécessaire. La suite des valeurs obtenues est :

$$\begin{array}{llllllll}
l_1 = 0 & l_2 = 0 & \cdots & l_{i-2} = 0 & l_{i-1} = 0 \\
l_1 = 0 & l_2 = 0 & \cdots & l_{i-2} = 0 & l_{i-1} = 1 \\
\vdots & \vdots & & \vdots & \vdots \\
l_1 = 0 & l_2 = 0 & \cdots & l_{i-2} = 0 & l_{i-1} = n_{i-1} - 1 \\
l_1 = 0 & l_2 = 0 & \cdots & l_{i-2} = 1 & l_{i-1} = 0 \\
\vdots & \vdots & & \vdots & \vdots \\
l_1 = n_1 - 1 & l_2 = n_2 - 1 & \cdots & l_{i-2} = n_{i-2} - 1 & l_{i-1} = n_{i-1} - 1
\end{array}$$

Ce même raisonnement peut s'appliquer au calcul des états correspondant aux automates à droite[5] de $\mathcal{A}^{(i)}$ (pour l'indice r de l'algorithme 4.1).

I.2.3. ALGORITHME DE MULTIPLICATION

L'algorithme 4.3 implémente les modifications nécessaires au traitement de termes avec éléments fonctionnels par rapport à l'algorithme 4.1 (**E-Sh** sans fonction). Ce nouvel algorithme est donc appelé **E-Sh** avec fonction. Les six opérations qui ont été ajoutées sont :

- ligne 1 : le parcours des facteurs normaux dans l'ordre de décomposition σ ;
- ligne 3 : l'initialisation du sous vecteur avec les états locaux des automates à gauche de la matrice $Q^{(i)}(\ldots)$;
- ligne 5 : l'initialisation du sous vecteur avec les états locaux des automates à droite de la matrice $Q^{(i)}(\ldots)$;
- ligne 12 : l'évaluation de la matrice $Q^{(i)}(\ldots)$ avec les arguments *calculés* (états locaux des automates) ;
- ligne 19 : l'incrément du sous vecteur avec les états locaux des automates à droite de la matrice $Q^{(i)}(\ldots)$;
- ligne 22 : l'incrément du sous vecteur avec les états locaux des automates à gauche de la matrice $Q^{(i)}(\ldots)$.

On peut remarquer que les sous vecteurs d'états locaux des automates à gauche et à droite de la matrice $Q^{(i)}(\ldots)$ sont redondants par rapport aux indices l et r, respectivement. Cette redondance a pour but d'éviter le calcul des sous vecteurs à chaque incrément de l et r,

[5] Nous appelons *à droite* d'une matrice d'indice i, toutes matrices d'un même terme qui ont des indices j supérieurs à i ($j \in [i+1..N]$).

qui est trop coûteux. De ce point de vue, les initialisations (lignes 3 et 5 de l'algorithme 4.3) et les incréments (lignes 22 et 19) sont équivalents aux initialisations et incréments sur l et r exécutés par les contrôles de boucles dans les lignes 4 et 6, respectivement.

Algorithme 4.3

```
1  for i = σ_1, σ_2, ..., σ_N
2  do  base = 0 ;
3      initialize   l_1 = 0, l_2 = 0, ..., l_{i-1} = 0 ;
4      for l = 0, 1, ..., nleft_i − 1          // boucle sur les portions
5      do  initialize   r_{i+1} = 0, r_{i+2} = 0, ..., r_N = 0 ;
6          for r = 0, 1, ..., nright_i − 1 // détail de la portion : boucle sur les z_in
7          do  index = base + r ;
8              for k = 0, 1, ..., n_i − 1 // extraction du z_in (sauts de nright_i)
9              do  z_in[k] = π̂[index] ;
10                 index = index + nright_i ;
11             end do
12             evaluate   Q^(i)(a_{l_1}^(1), ..., a_{r_N}^(N)) ; // évaluation de la matrice
13             multiply   z_out = z_in × Q^(i) ; // multiplication
14             index = base + r ;
15             for k = 0, 1, ..., n_i − 1 // stockage du résultat z_out (sauts de nright_i)
16             do  π̂[index] = z_out[k] ;
17                 index = index + nright_i ;
18             end do
19             next   r_{i+1}, r_{i+2}, ..., r_N ;
20         end do
21         base = base + ( nright_i × n_i ) ; // on passe à la portion suivante (saut dans π̂)
22         next   l_1, l_2, ..., l_{i-1} ;
23     end do
24 end do
```

Algorithme 4.3: **E-Sh** avec fonction

I.2.4. COMPLEXITÉ

L'algorithme 4.3 n'apporte pas de changement dans la taille des boucles par rapport à l'algorithme 4.1. La multiplication d'une tranche du vecteur $\hat{\pi}$ par les matrices $Q^{(i)}(\ldots)$ reste aussi identique. Ceci nous permet de dire que la complexité de l'algorithme 4.3 reste du même ordre que celle de l'algorithme 4.1. Les seuls coûts additionnels correspondent aux six opérations incluses. Le parcours des facteurs normaux dans l'ordre exprimé par σ est peu important au niveau de la complexité, car il ne fait qu'ajouter une indirection aux accès sur i. L'incrément du sous vecteur avec les états locaux des automates à droite de la matrice $Q^{(i)}(\ldots)$ et l'évaluation de la matrice $Q^{(i)}$ représentent un coût important, surtout

du fait que ces opérations doivent être faites dans la boucle la plus interne (boucle de 0 jusqu'à $nright_i - 1$).

Il est cependant difficile de donner une estimation théorique de la complexité lorsqu'il y a des dépendances fonctionnelles. Les cas pratiques sont très complexes, et seule une expérimentation numérique peut apporter des éléments d'information certains à propos des coûts d'exécution.

Les évaluations de matrices sont des opérations coûteuses, et des techniques de réordonnancement permettent d'en réduire le nombre en sortant ces évaluations de la boucle la plus interne de l'algorithme. Nous présenterons dans la section IV cette optimisation algorithmique.

I.3. CONCLUSION

L'algorithme du shuffle E-Sh utilise des vecteurs étendus, de la taille de l'espace d'états produit $\hat{S}$. De plus, nous effectuons dans l'algorithme 4.3 le calcul de tous les éléments du vecteur étendu, qui est de taille $|\hat{S}|$. Le remplissage d'une case du vecteur se fait à la ligne 16. Lorsque de nombreux états ne sont pas accessibles, nous effectuons donc beaucoup de calculs inutiles car la plupart des éléments du vecteur sont nul. On effectue en effet $|\hat{S}|$ multiplications d'une tranche de vecteur par une colonne de matrice.

Nous proposons dans les sections suivantes de nouvelles versions de cet algorithme qui utilisent des vecteurs réduits, de taille $|S|$ (S est l'espace des états accessibles du modèle).

[20] présente des algorithmes basés sur une telle représentation des vecteurs, mais l'algorithme du shuffle n'a pas pu être traité de cette façon. En effet, il se peut que des états non accessibles aient une probabilité non nulle dans des vecteurs intermédiaires. Ce phénomène sera détaillé dans la section II.1.

Les algorithmes de la section suivante se focalisent sur l'amélioration de la complexité en temps d'exécution, quitte à utiliser un peu de mémoire supplémentaire. L'utilisation mémoire sera optimisée dans les algorithmes de la section III.

II. AMÉLIORATION DE LA COMPLEXITÉ EN TEMPS D'EXÉCUTION : PR-Sh

Nous commençons par détailler les structures de données utilisées dans cet algorithme, et notamment nous cernons les instants où seuls les éléments correspondant à des états accessibles sont non nuls dans le vecteur en cours de calcul. Pour cela, nous introduisons les notions de *vecteur de probabilité* et de *vecteur intermédiaire*.

Ensuite, nous étudions l'intérêt de ne traiter que les états accessibles et le gain de calcul apporté. Pour cela, nous exposons les nouveaux algorithmes qui exploitent une structure de donnée creuse pour stocker les vecteurs de probabilité, et qui se focalisent sur une réduction du temps d'exécution.

Enfin, une étude de la complexité de ces algorithmes est effectuée.

II.1. STRUCTURES DE DONNÉES

Le fait de stocker uniquement les valeurs du vecteur qui correspondent à des états accessibles nous oblige à garder la trace des positions de ces éléments dans le vecteur correspondant dans $\hat{S}$. On suppose que l'ensemble des états accessibles S est connu. Il peut être calculé en appliquant un algorithme qui explore tous les états accessibles [67, 32], où bien en demandant à l'utilisateur de fournir une fonction qui représente l'ensemble des états accessibles [46].

Nous commençons par définir différents types de vecteurs que l'on est amené à rencontrer dans les algorithmes.

Définition 24 *Soit $\hat{\pi}$ un vecteur de taille $\hat{S}$. Chaque entrée du vecteur correspond à un état de $\hat{S}$. Alors,*

- *$pos_{\hat{\pi}}$ est l'ensemble des états ayant une entrée non nulle dans le vecteur $\hat{\pi}$.*
- *Un* **vecteur de probabilité** *est tel que $pos_{\hat{\pi}} \subseteq S$.*
- *Un* **vecteur intermédiaire** *est un vecteur qui n'est pas un vecteur de probabilité.*

Ainsi, les vecteurs de probabilité ont des entrées non nulles uniquement pour les états accessibles. En revanche, les vecteurs intermédiaires peuvent avoir une entrée non nulle pour un état non accessible.

Les vecteurs d'itération, que l'on est amené à multiplier par le descripteur, sont des vecteurs de probabilité. Nous détaillons maintenant les différentes étapes de la multiplication pour voir quand nous pouvons exploiter les propriétés des vecteurs de probabilité, et quand au contraire nous sommes en présence de vecteurs intermédiaires.

II.1.1. DÉTAIL DE LA MULTIPLICATION VECTEUR-DESCRIPTEUR

Le générateur Markovien Q correspondant à la chaîne de Markov associée à un réseau d'automates stochastiques est défini par la formule tensorielle suivante (cf formule 2.7) :

$$Q = \bigoplus_{i=1}^{N}{}_g \, Q_l^{(i)} + \sum_{e \in \mathcal{ES}} \left(\bigotimes_{i=1}^{N}{}_g \, Q_{e^+}^{(i)} + \bigotimes_{i=1}^{N}{}_g \, Q_{e^-}^{(i)} \right)$$

où N est le nombre d'automates du réseau, et $\mathcal{ES}$ est l'ensemble des identificateurs d'événements synchronisant.

La somme tensorielle correspond aux événements locaux du descripteur, et les produits tensoriels aux événements synchronisant. On étudiera les deux parties séparément ($Q = Q_{local} + Q_{synchro}$).

On veut effectuer la multiplication $\pi \times Q = \pi \times Q_{local} + \pi \times Q_{synchro}$, π étant un vecteur de probabilité (seuls les états accessibles ont une probabilité non nulle). La multiplication de π par Q revient à réaliser tous les événements (locaux et synchronisant) sur tous les automates en partant des états de probabilité non nulle (états de S). Le vecteur résultant $\pi \times Q$ est donc également un vecteur de probabilité, car par définition des états accessibles, on ne peut pas atteindre un état non accessible en partant d'un état accessible. Par contre, il se peut qu'on ait des vecteurs intermédiaires au cours du calcul. Pour cela, nous détaillons ce calcul.

Partie locale

Par définition de la somme tensorielle,

$$\pi \times Q_{local} = \pi \times \bigoplus_{i=1}^{N}{}_g \, Q_l^{(i)} = \sum_{i=1}^{N} \left[\pi \times (I_{nleft_i} \underset{g}{\otimes} Q_l^{(i)} \underset{g}{\otimes} I_{nright_i}) \right]$$

On doit calculer pour chaque automate $\pi \times (I_{nleft_i} \underset{g}{\otimes} Q_l^{(i)} \underset{g}{\otimes} I_{nright_i})$, ce qui revient à réaliser les événements locaux sur l'automate i. En partant des états accessibles (vecteur de probabilité π), on reste dans l'ensemble des états accessibles. En effet, soit l'événement local a un taux constant et toutes ses transitions mènent à un état accessible, soit l'événement a un taux fonctionnel dont certaines valeurs sont nulles si elles mènent à un état non accessible.

Chaque $\pi \times (I_{nleft_i} \underset{g}{\otimes} Q_l^{(i)} \underset{g}{\otimes} I_{nright_i})$ a donc des valeurs nulles pour tous les éléments non accessibles, et par définition c'est donc un vecteur de probabilité.

La somme de ces termes ($\pi \times Q_{local}$) est donc également un vecteur de probabilité (somme de vecteurs de probabilité).

Partie synchronisée

Pour la partie synchronisée,

$$\pi \times Q_{synchro} = \sum_{e \in \mathcal{ES}} \left(\pi \times \bigotimes_{i=1}^{N}{}_g \, Q_{e^+}^{(i)} + \pi \times \bigotimes_{i=1}^{N}{}_g \, Q_{e^-}^{(i)} \right)$$

On doit calculer pour chaque événement $e \in \mathcal{ES}$,

$$\pi' = \pi \times \bigotimes_{i=1}^{N}{}_g \, Q_{e^+}^{(i)} + \pi \times \bigotimes_{i=1}^{N}{}_g \, Q_{e^-}^{(i)}$$

ce qui revient à effectuer les transitions synchronisantes de l'événement e.

Lors du calcul de π', on est amené à faire des multiplications par des facteurs normaux (décomposition étudiée dans le cas de l'algorithme E-Sh). Lorsqu'on multiplie π par le ième facteur normal, on réalise l'événement synchronisant uniquement sur $\mathcal{A}^{(i)}$ (et pas sur les autres automates synchronisés avec $\mathcal{A}^{(i)}$). Des états non accessibles peuvent alors avoir une valeur non nulle dans le vecteur ainsi obtenu, ce sont des vecteurs intermédiaires. En revanche, lors de la multiplication par le dernier facteur normal, on a réalisé l'événement synchronisant sur tous les automates concernés, et on obtient donc un vecteur de probabilité.

Bilan

Lors de chaque multiplication par un facteur normal, on obtient un vecteur intermédiaire si on fait une multiplication dans la partie synchronisée qui n'est pas la multiplication par le dernier facteur normal. Dans tous les autres cas, et donc avant chaque addition, les vecteurs obtenus sont des vecteurs de probabilité.

Cette propriété peut être exploitée pour accélérer les algorithmes de multiplication par un facteur normal.

II.1.2. STRUCTURES UTILISÉES

Pour stocker un vecteur de probabilité, nous choisissons donc de garder en mémoire uniquement les entrées du vecteur qui correspondent aux états accessibles. Ceci nous oblige à stocker également les positions de ces éléments dans le vecteur correspondant dans $\hat{S}$.

Nous utilisons ainsi deux tableaux de taille $|S|$ pour stocker un vecteur de probabilité $\hat{\pi}$: le tableau $\hat{\pi}.vec$ contient les entrées correspondant aux états accessibles, et le tableau $\hat{\pi}.positions$ contient la place des états accessibles dans le vecteur correspondant dans $\hat{S}$. La figure 4.6 illustre les structures utilisées.

Les vecteurs intermédiaires peuvent être stockés de la même façon dans deux tableaux $\hat{\pi}.vec$ et $\hat{\pi}.positions$ de taille $|S|$ si $|pos_{\hat{\pi}}| \leq |S|$. Dans ce cas, les positions ne correspondent plus aux états accessibles, mais elles correspondent aux états de $pos_{\hat{\pi}}$.

On peut remarquer que, la plupart du temps, $|pos_{\hat{\pi}}| \leq |S|$ car les vecteurs intermédiaires sont obtenus via une multiplication par une matrice de synchronisation, et ces matrices sont généralement très creuses. Ces vecteurs contiennent donc de nombreux éléments non nuls.

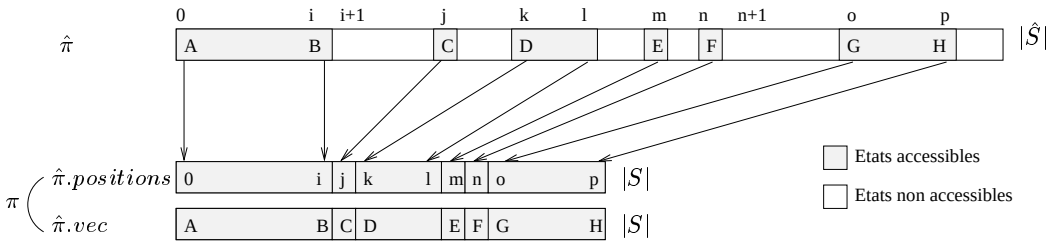


FIG. 4.6 – Illustration des structures utilisées pour les algorithmes exploitant le creux

Cependant, il peut arriver que $|pos_{\hat{\pi}}| > |S|$. Dans ce cas, il devient nécessaire de réallouer dynamiquement de la mémoire pour pouvoir stocker le vecteur. Ceci arrive cependant très rarement d'après notre expérience.

II.2. ALGORITHME PR-SH

Maintenant que nous avons détaillé les structures de données que nous allons utiliser, nous proposons un nouvel algorithme de multiplication des facteurs normaux qui exploite le fait que le vecteur obtenu par la multiplication d'un vecteur de probabilité par le descripteur est toujours un vecteur de probabilité. Nous pouvons ainsi réduire le nombre de multiplications de $|\hat{S}|$ à $|S|$, en calculant seulement les éléments du résultat qui correspondent à des états accessibles (donc, leur valeur est susceptible d'être non nulle). Lorsqu'on est confronté à un vecteur intermédiaire, on est cependant obligé d'effectuer tous les calculs pour trouver quels éléments du vecteur résultat sont non nuls.

Ainsi, les vecteurs (notés π) que l'on manipule sont de taille $|S|$. En comparaison, les vecteurs $\hat{\pi}$ utilisés dans l'algorithme E-Sh étaient systématiquement de taille $|\hat{S}|$.

Nous voulons appliquer le même principe que l'algorithme E-Sh exposé précédemment. Q est décomposé en facteurs normaux, et on détaille le calcul d'un vecteur par un facteur normal i ($i \in [1..N]$) :

$$\pi' = \pi \times I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$$

II.2.1. MULTIPLICATION PAR UN FACTEUR NORMAL

Le format d'un facteur normal ne change pas par rapport à l'algorithme E-Sh, la notion de portion reste donc la même.

La découpe de π en portions p_l ($l = 0 \,..\, nleft_i - 1$) s'effectue en regardant le tableau $\pi.positions$, et en sélectionnant les éléments qui correspondent à la portion. Une portion étant un ensemble d'éléments consécutifs, un unique balayage du vecteur permet d'en déterminer les limites. Chaque portion p_l est représentée par des sous-tableaux de $\pi.positions$ et

de $\pi.vec$, notés respectivement $p_l.positions$ et $p_l.vec$. La figure 4.7 illustre ce découpage.

On travaille ensuite en effectuant une boucle sur les portions, comme précédemment.

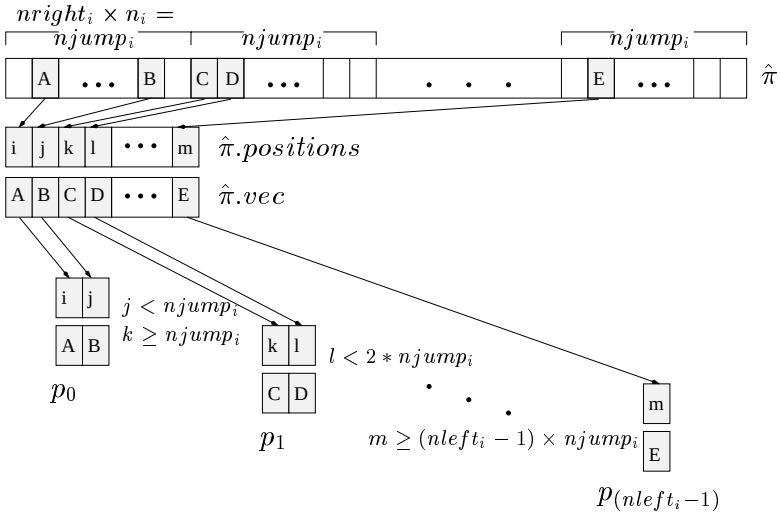


FIG. 4.7 – Découpe du vecteur π en portions p_l

II.2.2. DÉTAIL D'UNE PORTION

Soit $l \in [0 \,..\, nleft_i - 1]$. On s'intéresse maintenant au traitement de la portion p_l.

La principale difficulté d'application de E-Sh avec la nouvelle structure de vecteurs réside dans l'extraction des tranches de vecteur z_{in} à partir d'une portion $p_l.vec$ et $p_l.positions$.

En effet, E-Sh utilise une technique de sauts pour extraire les tranches de vecteur.

Lorsque le vecteur est stocké dans une structure de la taille de S, on ne peut plus effectuer de sauts. Une méthode similaire peut cependant être adoptée en parcourant le vecteur pour chaque extraction d'un z_{in}, et en récupérant les éléments non nuls qui appartiennent à ce z_{in}. A chaque itération de la boucle sur $nright_i$ (ligne 6 de l'algorithme 4.3), on parcourt la portion de vecteur et on stocke les éléments non nuls correspondant au z_n.

Pour éviter les nombreux parcours de vecteur engendrés par cette approche (1 parcours par z_{in} à extraire, soit $nright_i$ parcours par portion), on décide de récupérer tous les z_n pour la portion que l'on traite lors d'un seul parcours. Le coût d'extraction des z_n est alors divisé par $nright_i$.

On décide de stocker tous les z_{in} correspondant à une portion de vecteur p_l dans un tableau. Les z_{in} sont similaires à ceux employés dans l'algorithme E-Sh ; nous les stockons ainsi dans un format étendu. Ainsi, pour chaque portion p_l (stockée de façon réduite), on construit un vecteur étendu qui contient la suite des z_n, dans l'ordre adéquat pour ce facteur normal.

Chaque élément de p_l appartient donc à un z_{in} donné, et il est à une certaine place dans ce z_{in}. Le numéro de z_{in} indique le z_{in} auquel appartient un élément. La numérotation va de

0 à $nright_i - 1$, et correspond à la numérotation introduite dans la section I.1.2 (algorithme E-Sh). La place dans le z_{in} indique tout simplement la place d'un élément dans un z_{in} donné.

La structure qui héberge les z_{in} est un tableau à deux dimensions. Ainsi, $z_{in}[r]$ représente le z_{in} n° r ; c'est un tableau de taille n_i contenant tous les éléments du z_{in} n° r. $z_{in}[r][k]$ représente donc l'élément situé à la place k dans le z_{in} n° r. z_{in} est un tableau de taille $nright_i \times n_i$.

Pour extraire les z_{in}, on effectue une division entière des valeurs du tableau $p_l.positions$, ce qui nous donne, pour j variant de 0 à $|S|$, le numéro du z_{in} auquel appartient l'élément correspondant ($p_l.positions[j]$ mod $nright_i$), ainsi que sa place dans ce z_{in} ($p_l.positions[j]$ div $nright_i$). Une division entière par $nright_i$ revient en effet à récupérer des éléments espacés de $nright_i$ dans le vecteur étendu. La figure 4.8 illustre cette technique d'extraction des z_{in}.

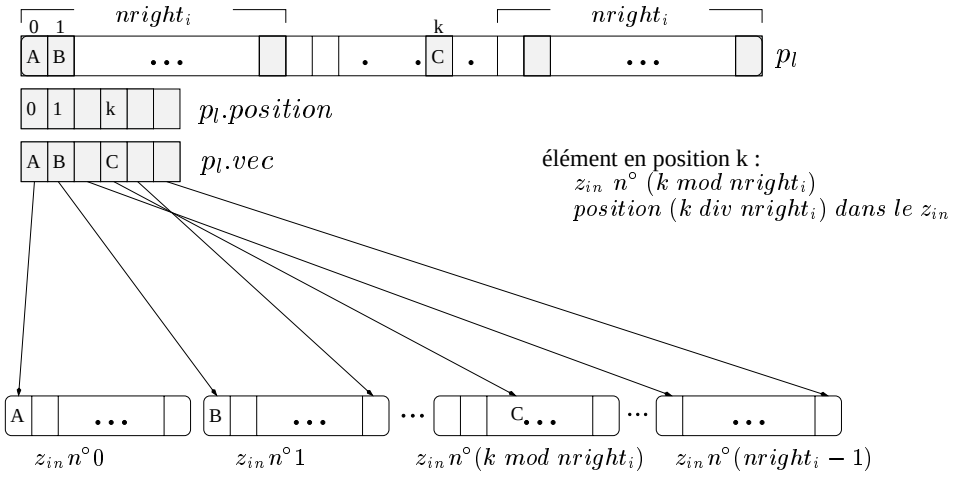


FIG. 4.8 – Extraction de tous les z_{in} pour une portion de vecteur réduit p_l

Une fois que l'on dispose des z_{in}, il reste à effectuer les multiplications $z_{in} \times q_{*,k}$ (se reporter à la présentation de l'algorithme E-Sh, figure 4.3) puis à stocker les résultats à la bonne place dans le vecteur résultat $\pi' = \pi \times I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$ (format réduit : $\pi'.vec$ et $\pi'.positions$). Il faut aussi effectuer les évaluations des matrices dans le cas de matrices fonctionnelles.

On distingue alors le cas où π' est un vecteur de probabilité du cas où π' est un vecteur intermédiaire. Ce dernier cas a lieu lors de la multiplication par un facteur normal correspondant à un événement synchronisant, si ce n'est pas le dernier facteur normal.

II.2.3. ALGORITHME : CAS OÙ π' EST UN VECTEUR DE PROBABILITÉ

On suppose disposer maintenant de tous les z_{in} (pour une portion). On peut donc accéder directement à un z_{in} donné.

Le fait que π' soit un vecteur de probabilité nous indique quels éléments sont susceptibles d'être non nuls dans π' : ce sont les éléments correspondant aux états accessibles. On se contente donc de calculer la valeur correspondant à ces états. Pour cela, on parcourt le vecteur π' (dont les valeurs du tableau $\pi'.positions$ correspondent aux états accessibles) et on calcule la probabilité pour chaque état accessible, comme indiqué ci-après.

On a vu lors de la présentation de l'algorithme E-Sh (cf figure 4.3) que le calcul d'un élément du vecteur résultat se fait en multipliant un z_{in} donné par une colonne de la matrice $Q^{(i)}$.

Pour obtenir le numéro du z_{in} et la colonne de la matrice permettant d'effectuer la multiplication pour un élément donné du vecteur résultat, le schéma est similaire à celui correspondant à l'extraction des z_{in}. En effet, les éléments correspondant à la multiplication par le même z_{in} sont espacés de $nright_i$ dans le vecteur étendu. Si on désire calculer la valeur de l'élément placé en position k, on commence par faire une division entière de k par $nright_i$. Le reste de la division nous donne le z_{in} à utiliser pour la multiplication, et le quotient nous donne la colonne de la matrice. On effectue alors la multiplication, comme l'illustre la figure 4.9.

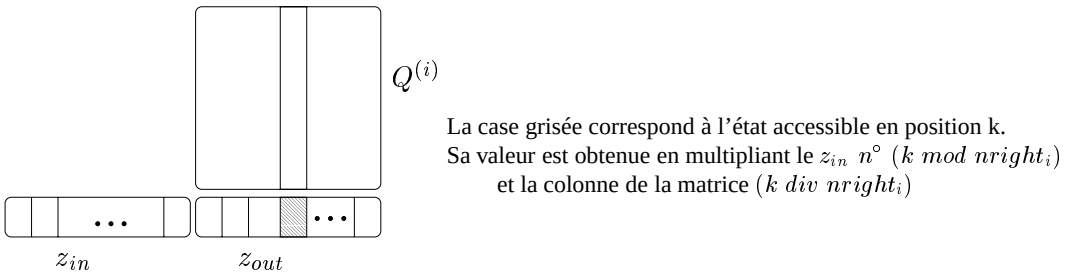


FIG. 4.9 – Calcul d'une portion de π', vecteur de probabilité

L'algorithme 4.4 effectue la multiplication $\pi' = \pi \times (I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i})$. Nous appelons cet algorithme **PR-Sh**, pour *partially reduced*, car les vecteurs π et π' sont stockés dans un format réduit mais l'algorithme utilise des structures intermédiaires en format étendu (les z_{in}). L'algorithme correspond au cas sans fonction, et π' est un vecteur de probabilité.

Les boucles **while** (lignes 8 et 14 de l'algorithme 4.4) servent à limiter notre étude par portion. La portion l correspond en effet aux éléments du vecteur situés entre les positions $l \times njump_i$ et $((l+1) \times njump_i) - 1$. Les variables j et $j2$ assurent la progression dans v et dans le vecteur résultat w. On remarquera que l'on n'effectue pas la division de la position d'un élément par $nright_i$, mais que l'on commence par soustraire $l \times njump_i$ à la position. En effet, $l \times njump_i$ correspond à la position du début de la portion l. Pour obtenir un quotient compris entre 0 et $n_i - 1$, cette soustraction est nécessaire. Sinon, le quotient sera compris entre $l \times n_i$ et $((l+1) \times n_i) - 1$. En théorie, la soustraction n'est nécessaire que pour le calcul du quotient, mais en pratique on effectue les deux calculs en même temps. C'est pourquoi nous effectuons la soustraction également pour le calcul du reste.

Algorithme 4.4

```
1   j = 0 ; j2 = 0 ;
2   for l = 0, 1, ..., nleft_i − 1          // boucle sur les portions
3   do  for r = 0, 1, ..., nright_i − 1          // initialisation des z_in
4       do  for k = 0, 1, ..., n_i − 1
5           do  z_in[r][k] = 0 ;
6           end do
7       end do
        // Remplissage des z_in
8       while   (j < |S|) && (π.positions[j] < (l + 1) × njump_i)
9       do  re = (π.positions[j] − l × njump_i) mod nright_i ;
10          qo = (π.positions[j] − l × njump_i) quo nright_i ;
11          z_in[re][qo] = π.vec[j] ;
12          j = j + 1 ;
13      end do
        // Calcul du résultat
14      while   (j2 < |S|) && (π'.positions[j2] < (l + 1) × njump_i)
15      do  re = (π'.positions[j2] − l × njump_i) mod nright_i ;
16          qo = (π'.positions[j2] − l × njump_i) quo nright_i ;
            // Q^(i)[qo] représente la colonne qo de la matrice Q^(i)
17          π'.vec[j2] = z_in[re] × Q^(i)[qo] ;
18          j2 = j2 + 1 ;
19      end do
20  end do
```

Algorithme 4.4: **PR-Sh** sans fonction - π' est un vecteur de probabilité

Traitement des dépendances fonctionnelles

Un problème se pose pour évaluer les éléments fonctionnels, car on ne peut plus procéder en parcourant $\pi.positions$ de façon linéaire. Si on procédait ainsi, il faudrait réévaluer les matrices fonctionnelles pour chaque produit d'une colonne de $Q^{(i)}$ par un z_{in}. Or, l'évaluation des éléments fonctionnels est très coûteux. On désire donc procéder comme dans l'algorithme 4.3 (**E-Sh** avec fonction), qui implémente une solution moins coûteuse. En effet, dans cet algorithme, on change d'état (argument des fonctions) en procédant par incrément. Le changement d'état se fait lorsqu'on change de z_{in} (boucle ligne 6 à 20 de l'algorithme 4.3). Pour un z_{in} donné, une seule évaluation de fonction est nécessaire.

Dans le cas des vecteurs de probabilité réduits, il faut donc procéder autrement que dans l'algorithme 4.4, qui effectue un parcours linéaire. Nous devons maintenant traiter les z_{in} dans le même ordre que dans l'algorithme 4.3.

L'algorithme 4.5 effectue la multiplication avec les évaluations de fonction dans la boucle la plus interne. Nous détaillons les différentes étapes de cet algorithme.

L'initialisation des z_{in} et leur extraction a lieu de la même façon que pour l'algorithme sans évaluation de fonction (algorithme 4.4, lignes 1 à 12). On doit rajouter pour les fonctions l'initialisation des sous vecteurs avec les états locaux des automates, ainsi que l'initialisation de quelques tableaux détaillés ci-dessous.

Ces tableaux servent à stocker les informations sur les calculs à effectuer lorsqu'on n'effectue plus un parcours linéaire. En effet, pour diminuer le nombre de calculs, on conserve en mémoire les éléments susceptibles d'être non nuls, ainsi on pourra effectuer le calcul uniquement pour ces éléments, contrairement à E-Sh qui faisait systématiquement tous les calculs. Un algorithme n'utilisant pas de tableaux devrait faire $nright_i$ parcours de la portion de π' pour se limiter aux évaluations nécessaires, et au calcul d'éléments susceptibles d'être non nuls (un parcours par z_{in} à traiter). Or, un unique parcours de la portion de π' nous indique tous les éléments susceptibles d'être non nuls, mais ces éléments ne sont pas donnés dans le bon ordre. On les stocke donc pour pouvoir faire les calculs au moment voulu. Dans le cas de l'algorithme 4.4, un simple parcours nous indiquait, les uns après les autres, les éléments à calculer. Il n'y avait donc pas besoin de tableaux, car on traitait les éléments dans un ordre linéaire.

Le parcours est effectué lignes 13 à 20 de l'algorithme 4.5. Nous stockons les informations intermédiaires dans les tableaux suivants :

- le tableau **used** détermine les z_{out} qui contiennent au moins un état accessible. Les z_{out} sont numérotés de la même façon que les z_{in} : $z_{out}[r] = z_{in}[r] \times Q^{(i)}(\dots)$ avec $r \in [0..nright_i - 1]$. **used** est un tableau de booléens de taille $nright_i$. $used[r]$ vaut la valeur $true$ si et seulement si le z_{out} n° r contient au moins un élément qui correspond à un état accessible (un élément susceptible d'être non nul). A chaque changement de portion, on initialise donc toutes les valeurs de ce tableau à $false$.
- le tableau **indice** est un compteur pour savoir combien d'états accessibles sont présents dans chaque z_{out}. Ainsi, le z_{out} n° r contient $indice[r]$ états accessibles.
- le tableau **useful** indique la place dans le z_{out} d'un élément à calculer. On ne note que les places correspondant aux états accessibles, et on les numérote de 1 à $indice[r]$ pour le z_{out} n° r. $useful[r][k]$ correspond à la place dans le z_{out} n° r du k^{ieme} état accessible de ce z_{out}.

Algorithme 4.5

```
1   j = 0 ; j2 = 0 ;
2   initialize   l_1 = 1, l_2 = 1, ..., l_{i-1} = 1 ;
3   for l = 0, 1, ..., nleft_i - 1          // boucle sur les portions
4   do  initialisation de z_in à 0
5       initialisation de indice à 0
6       initialisation de used à false
        // Remplissage des z_in
7       while   (j < |S|) && (π.positions[j] < (l + 1) × njump_i)
8       do  re = (π.positions[j] − l × njump_i) mod nright_i ;
9           qo = (π.positions[j] − l × njump_i) quo nright_i ;
10          z_in[re][qo] = π.vec[j] ;
11          j = j + 1 ;
12      end do
        // Informations intermédiaires
13      while   (j2 < |S|) && (π'.positions[j2] < (l + 1) × njump_i)
14      do  re = (π'.positions[j2] − l × njump_i) mod nright_i ;
15          qo = (π'.positions[j2] − l × njump_i) quo nright_i ;
16          used[re] = true ;            // Quels z_out devront être calculés
17          useful[re][indice[re]] = qo ;   // Pour un z_out donné, les places utilisées
            // Pour un z_out donné, la place dans le vecteur où stocker le résultat
18          place[re][indice[re]] = j2 ;
19          indice[re] = indice[re] + 1 ; j2 = j2 + 1 ;
20      end do
21      initialize   r_{i+1} = 1, r_{i+2} = 1, ..., r_N = 1 ;
22      for r = 0, 1, ..., nright_i - 1          // Calcul du résultat
23      do  if   (used[r])
24              evaluate   Q^(i)(a^(1)_{l_1}, ..., a^(N)_{r_N}) ; // Evaluation uniquement si nécessaire
25              for k = 0, 1, ..., indice[r] - 1
                // Multiplication par une colonne de la matrice
26              do  π'.vec[place[r][k]] = z_in[r] × Q^(i)[useful[r][k]] ;
27              end do
28          end if
29          next   r_{i+1}, r_{i+2}, ..., r_N ;
30      end do
31      next   l_1, l_2, ..., l_{i-1} ;
32  end do
```

Algorithme 4.5: **PR-Sh** avec fonction - π' est un vecteur de probabilité

– le tableau **place** donne la place dans le vecteur résultat des éléments des z_{out}. Une fois une valeur calculée, il faut stocker le résultat dans le vecteur. L'élément numéroté k dans le z_{out} n° r sera donc placé à $place[r][k]$ dans le vecteur résultat.

Ensuite, on calcule les éléments non nuls comme précédemment, mais pas dans le même ordre (on calcule tous ceux correspondant à un même z_{in} à la suite). Pour cela, on effectue une boucle sur $nright_i$ (ligne 22), et on effectue l'évaluation de la matrice uniquement si elle est nécessaire (ligne 24). L'évaluation doit être faite si on doit calculer au moins un élément du z_{out} correspondant, donc si la valeur de $used$ correspondante vaut $true$.

Enfin, on calcule chaque élément correspondant à un état accessible de ce z_{out} n° r, en faisant une boucle de 0 à $indice[r] - 1$ (ligne 25). Le tableau $place$ nous indique où stocker le résultat dans le vecteur, et le tableau $useful$ nous indique la colonne de la matrice à employer dans la multiplication. Le calcul ligne 26 est similaire à celui effectué en ligne 17 de l'algorithme 4.4.

II.2.4. ALGORITHME : CAS OÙ π' EST UN VECTEUR INTERMÉDIAIRE

Lors de la multiplication d'un vecteur π par un produit tensoriel $\bigotimes_{g_{i=1}}^{N} Q_e^{(i)}$ correspondant au traitement d'un événement synchronisant e, on peut obtenir des vecteurs intermédiaires, qui comportent des entrées non nulles pour des états non accessibles. On ne peut donc plus effectuer uniquement les calculs d'un z_{in} par une colonne de matrice qui donnent un résultat correspondant à un état accessible, car on ne sait pas où seront les valeurs non nulles dans le vecteur résultat. On est obligé dans ce cas d'effectuer les produits $z_{in} \times Q_e^{(i)}$, et de ne garder que les éléments non nuls en sortie. On doit alors mettre à jour le tableau $\pi'.positions$ lorsqu'on remplit le vecteur résultat π' avec les éléments non nuls obtenus, puisqu'on ne connaît pas forcément leurs positions (ce ne sont pas obligatoirement des états accessibles).

Dans la plupart des cas, le nombre d'états avec une probabilité non nulle en sortie est inférieur ou égal à $|S|$, car les matrices de synchronisation sont très creuses. On peut donc utiliser la même taille de tableaux. Cependant, il arrive (rarement d'après notre expérience, et sur des exemples pas très réalistes) que le nombre d'états avec une entrée non nulle augmente. Dans ce cas, il est nécessaire de réallouer dynamiquement de la place mémoire pour stocker le vecteur (les tableaux $\pi'.vec$ et $\pi'.positions$ ne sont plus de taille suffisante).

Les algorithmes qui précèdent (4.4 et 4.5) lisent séparément $\pi.positions$ et $\pi'.positions$. En effet, lorsque π' est un vecteur de probabilité, les valeurs de $\pi'.positions$ sont déjà initialisées aux positions des états accessibles. Par contre, il se peut que les entrées de $\pi.positions$ soient différentes, car π peut être un vecteur intermédiaire.

Lorsqu'on sait que π' est un vecteur intermédiaire, on calcule les entrées de $\pi'.positions$ lors de la multiplication vecteur-matrice. On ne suppose plus que $\pi'.positions$ est connu comme dans le cas précédent.

La principale différence avec l'algorithme 4.5 est qu'on n'effectue pas la recherche des éléments à calculer (lignes 13 à 20), tout simplement parcequ'on est obligé de tout calculer dans ce cas. On ne fait donc plus la multiplication d'un z_{in} par une colonne de matrice, mais

la multiplication du z_{in} par la matrice entière, et on stocke tous les éléments non nuls sous forme de couples (position, valeur). Lorsqu'on a fini les multiplications pour une portion, on stocke les éléments non nuls obtenus dans le vecteur résultat, en mettant à jour les tableaux $\pi'.positions$ et $\pi'.vec$.

L'algorithme 4.6 effectue la multiplication dans le cas où π' est un vecteur intermédiaire, avec les évaluations de fonction. Nous traitons ici les z_{in} dans le même ordre que dans l'algorithme 4.3 ; les évaluations se font donc de la même façon. La multiplication ligne 14 stocke dans la structure z_{out} les éléments non nuls obtenus lors du calcul $z_{in} \times Q^{(i)}$. Après avoir classé ces éléments par position croissante (ligne 17), on peut stocker les résultats dans le vecteur solution (lignes 18 à 22). On n'utilisera pas forcément tout le tableau $\pi.vec$, d'où l'intérêt de mettre une position égale à -1 lorsqu'on a stocké tous les éléments non nuls. On sait ainsi quelle partie du tableau est utilisée.

II.3. COMPLEXITÉ

Nous terminons cette section par une étude de la complexité des algorithmes **PR-Sh**, et par une comparaison théorique avec l'algorithme **E-Sh**. Dans un premier temps nous regarderons le nombre d'opérations effectuées, puis la place mémoire occupée.

La complexité du produit d'un vecteur par un terme produit tensoriel classique est obtenue en observant le nombre de multiplications d'une tranche de vecteur par une colonne de matrice. Dans le cas de l'algorithme **E-Sh**, on effectue $|\hat{S}|$ produits, alors qu'on n'en effectue plus que $|S|$ pour l'algorithme **PR-Sh** dans le cas où le vecteur résultat est un vecteur de probabilité. Lorsqu'on obtient un vecteur intermédiaire, nous ne pouvons pas exploiter les propriétés qui nous permettent de réduire le nombre de calculs. Le gain par rapport à l'algorithme **E-Sh** est donc nul. Cet algorithme est donc intéressant au niveau du nombre de calcul uniquement lorsqu'on obtient un vecteur de probabilité.

Dans le cas de matrices fonctionnelles, la complexité reste du même ordre (le nombre de calculs ne change pas), mais l'évaluation de la matrice représente un coût important. Nous diminuons le nombre d'évaluations par rapport à l'algorithme **E-Sh** si certaines multiplications vecteur-matrice donnent un résultat nul. S'il y a beaucoup d'états non accessibles, cette diminution du nombre d'évaluations est significative.

Au niveau de la place mémoire occupée par **PR-Sh**, on constate que les vecteurs sont stockés dans un format réduit, ce qui permet un gain de mémoire par rapport à l'algorithme du shuffle. Les vecteurs réduits sont en effet de taille $2 \times |S|$ (deux tableaux de taille $|S|$), alors que les vecteurs étendus sont de taille $|\hat{S}|$. Cependant, les structures intermédiaires qui servent à (**PR-Sh**) sont stockées dans un format étendu. En effet, nous utilisons des tableaux de taille $nright_i \times n_i$ pour stocker les z_{in} et les éventuelles informations intermédiaires. Or, lorsqu'on traite le premier facteur normal, $nleft_i = 1$. Par définition, $nright_i \times n_i \times nleft_i = |\hat{S}|$. On utilise ainsi un tableau de taille $|\hat{S}|$. Si ce tableau ne tient pas en mémoire, l'algorithme ne peut donc pas être appliqué. L'algorithme du shuffle classique (**E-Sh**) stocke le vecteur dans une structure de taille $|\hat{S}|$. Le gain en mémoire du nouvel algorithme est donc nul.

Algorithme 4.6

```
1   j = 0 ; j2 = 0 ;
2   initialize   l_1 = 1, l_2 = 1, ..., l_{i-1} = 1 ;
3   for l = 0, 1, ..., nleft_i - 1          // boucle sur les portions
4   do  initialiser les z_in à 0 et effacer z_out ;
        // Remplissage des z_in
5       while   (j < |S|) && (π.positions[j] < (l + 1) × njump_i)
6       do  re = (π.positions[j] − l × njump_i) mod nright_i ;
7           qo = (π.positions[j] − l × njump_i) quo nright_i ;
8           z_in[re][qo] = π.vec[j] ;
9           j = j + 1 ;
10      end do
11      initialize   r_{i+1} = 1, r_{i+2} = 1, ..., r_N = 1 ;
12      for r = 1, 2, ..., nright_i          // Calcul du résultat
13      do  evaluate   Q^(i)(a_{l_1}^(1), ..., a_{r_N}^(N)) ;
14          multiply   z_out = z_in × Q^(i) ;
15          next   r_{i+1}, r_{i+2}, ..., r_N ;
16      end do
17      classer z_out par ordre de positions croissantes ;
18      while   ((pos, val) ∈ z_out)          // Stockage du résultat de la portion
19      do  π'.positions[j2] = pos + l × njump_i ;
20          π'.vec[j2] = val ;
21          j2 = j2 + 1 ;
22      end do
23      if   (j2 ≥ |S|)
24          Prévenir l'utilisateur du débordement ou réallocation dynamique.
25      end if
26      next   l_1, l_2, ..., l_{i-1} ;
27  end do
    // Si nécessaire, met un marqueur pour savoir où s'arrête le tableau utilisé
28  if   (j2 < |S|)
29      π'.positions[j2] = −1
30  end if
```

Algorithme 4.6: **PR-Sh** avec fonction - π' est un vecteur intermédiaire

Dans la section suivante, nous proposons une optimisation de cet algorithme au niveau de la place mémoire. Le nouvel algorithme totalement réduit $FR-Sh$ (*fully reduced shuffle*) n'utilise plus aucune structure de taille $|\hat{S}|$.

III. OPTIMISATION DE LA PLACE MÉMOIRE UTILISÉE : FR-Sh

Les algorithmes de la section précédente (**PR-Sh**) sont optimisés pour gagner du temps à l'exécution : on ne calcule que les éléments du vecteur résultat qui seront non nuls lorsqu'on peut le prévoir. Cependant, le nombre de calculs reste élevé lorsque le vecteur résultat est un vecteur intermédiaire, et le gain en mémoire par rapport à l'algorithme **E-Sh** est nul, car nous utilisons encore des structures de taille $|\hat{S}|$.

Nous présentons maintenant de nouveaux algorithmes permettant de n'utiliser aucune structure de taille $|\hat{S}|$, quitte à perdre un peu de temps à l'exécution dans le cas où le vecteur résultat est un vecteur de probabilité.

Nous présenterons tout d'abord les structures de données que l'on est amené à utiliser dans ce nouvel algorithme. Ensuite, nous détaillerons l'algorithme FR-Sh (*fully reduced shuffle*), et enfin donnerons une étude de complexité de cet algorithme.

III.1. STRUCTURES DE DONNÉES

La principale difficulté pour appliquer les idées de l'algorithme E-Sh en n'utilisant que des vecteurs réduits (et sans structures intermédiaires étendues comme dans PR-Sh) réside dans l'extraction des tranches de vecteur z_{in} à partir des portions. En effet, E-Sh procédait par sauts, et cette technique n'est plus possible lorsque le vecteur est réduit.

Une méthode en quelque sorte similaire peut cependant être adoptée, qui consiste à traverser la portion de vecteur que l'on traite et à en extraire tous les z_{in}. Contrairement à PR-Sh, les z_{in} seront cependant stockés dans une structure réduite.

Nous utilisons pour cela la librairie STL (Standard Template Library), qui contient des *conteneurs* pour stocker et organiser un ensemble d'objets, des *itérateurs* pour pouvoir balayer ces conteneurs et accéder à leurs éléments, et des *algorithmes génériques* qui agissent sur les conteneurs (algorithmes de tri, de recherche, ...) [79].

La structure intermédiaire consiste en un ensemble de triplets (num, place, index), chaque triplet correspondant à un élément du vecteur π :

- **num** représente le numéro du z_{in} qui contient cet élément ;
- **place** représente la place de l'élément dans le z_{in} ;
- **index** représente l'index (la position) de l'élément dans le vecteur réduit ($1 \leq index \leq |S|$). Il nous permet de trouver la valeur de l'élément : $\pi.vec[index]$.

Nous détaillons maintenant les différentes étapes de la multiplication

$$\pi' = \pi \times (I_{nleft_i} \underset{g}{\otimes} Q_l^{(i)} \underset{g}{\otimes} I_{nright_i})$$

pour l'algorithme FR-Sh. Comme pour la présentation de PR-Sh, on séparera le cas où π' est un vecteur de probabilité du cas où c'est un vecteur intermédiaire pour optimiser les performances du nouvel algorithme.

III.2. ALGORITHME FR-SH

Il faut déjà remarquer qu'une portion est un ensemble d'éléments consécutifs et qu'un simple parcours du vecteur π nous permet d'en déterminer les limites. Ainsi, pour $l \in [0..nleft_i - 1]$, la portion p_l contient les éléments avec un index x tel que $n_i \times nright_i \times l \leq \pi.positions(x) < n_i \times nright_i \times (l+1)$.

On traite les portions de façon séquentielle (mais on pourrait envisager de les traiter en parallèle, chaque traitement étant indépendant des autres portions). Ainsi, pour chaque portion, on commence par récupérer les informations nécessaires pour une exécution efficace du reste de l'algorithme. L'ensemble de triplets *infozin* contient les informations sur les z_{in} de la portion courante, tandis que *infozout* donne des informations sur les z_{out} lorsqu'on en a (cas où π' est un vecteur de probabilité).

Calcul de $r_l = p_l \times Q^{(i)} \otimes I_{nright_i}$:

1. Remplissage de *infozin* :
On commence par effectuer un parcours de $p_l.positions$, pour construire un ensemble de triplets appelé *infozin*. Pour chaque élément, on obtient num et $place$ en effectuant une division entière de $p_l.positions[index]$ par $nright_i$:
$num = p_l.positions[index] \ mod \ nright_i$ et
$place = p_l.positions[index] \ div \ nright_i$.
Une division entière par $nright_i$ revient à aller chercher les éléments espacés de $nright_i$ dans le vecteur étendu.

2. Remplissage de *infozout* :
Lorsque π' est un vecteur de probabilité, le tableau $\pi'.positions$ est initialisé avec les positions des états de S. Cela signifie que l'on peut effectuer un parcours de la portion $r_l.positions$, pour construire un ensemble *infozout* similaire à *infozin*, mais qui prend en compte les positions des éléments non nuls dans la portion résultat r_l (elles peuvent être différentes de celles de p_l dans le cas où π est un vecteur intermédiaire). Ainsi, les champs num et $place$ représentent respectivement, pour chaque élément de η_l, le numéro de z_{out} et la place de l'élément dans le z_{out}.
Lorsque π' est un vecteur intermédiaire, on ne dispose d'aucune information sur la place des éléments non nuls dans ce vecteur, donc il n'y a pas de *infozout*.

3. Tri de *infozin* et *infozout* :
On doit trier[6] l'ensemble des triplets selon les num croissants, pour que tous les éléments d'un même z_{in} (ou z_{out}) se retrouvent côte à côte.

4. Traitement d'un z_{in} :

a. Extraction du z_{in} : Les éléments d'un même z_{in} sont maintenant placés côte à côte, et un unique parcours de *infozin* nous permet de récupérer les z_{in} les uns après les autres, comme le montre la figure 4.10.
Lorsque π' est un vecteur de probabilité, on n'extrait le z_{in} que si au moins un des états du z_{out} correspondant est accessible. Ceci s'obtient étalement par un unique parcours de *infozout*, car les éléments du même z_{out} sont maintenant placés côte à côte. Ainsi, nous n'avons pas forcément besoin d'extraire tous les z_{in}, comme c'était le cas dans E-Sh.

b. Multiplication :
Lorsque π' est un vecteur de probabilité, *infozout* nous indique quels éléments doivent être calculés, et on effectue pour ces éléments la multiplication du z_{in} par la colonne de matrice correspondante. Dans ce cas, on n'a pas besoin d'effectuer toutes les multiplications comme dans E-Sh.
D'un autre côté, lorsque π' est un vecteur intermédiaire, on ne connaît pas a priori les positions des éléments non nuls, donc on doit effectuer la multiplication du z_{in} par la matrice entière, et on stocke tous les éléments non nuls du résultat sous forme d'un couple (position, valeur) (comme dans l'algorithme PR-Sh).

5. Stockage du résultat :
Lorsque π' est un vecteur de probabilité, on effectue une multiplication pour chaque entrée du vecteur correspondant à un état accessible, et l'information contenue dans *infozout* nous indique où stocker la valeur obtenue ($\pi'.vec[index]$). Le stockage du résultat peut donc avoir lieu au fur et à mesure des multiplications.
D'un autre côté, lorsque π' est un vecteur intermédiaire, on doit attendre la fin des calculs correspondant à chaque z_{in}, puis trier les éléments non nuls obtenus par positions croissantes. Ensuite, un parcours linéaire de ces éléments est suffisant pour remplir les tableaux $\pi'.positions$ et $\pi'.vec$[7].

Lorsque tous les z_{in} correspondant à une portion p_l ont été traités, la portion suivante est traitée.

[6] Le tri de STL utilisé est *introsort*, une variante du tri rapide *quicksort*, qui offre une complexité en $O(N \log N)$ dans le cas le pire.

[7] C'est à ce moment que l'on peut avoir besoin d'une réallocation dynamique, si le nombre d'éléments non nuls est supérieur à |S|.

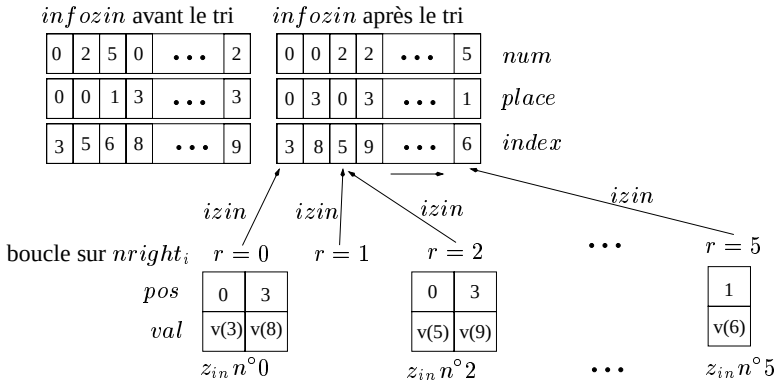


FIG. 4.10 – Extraction des z_{in} à partir de $infozin$

Les algorithmes 4.7 et 4.8 correspondent respectivement au cas où π' est un vecteur de probabilité et au cas où c'est un vecteur intermédiaire, avec les évaluations de fonction dans la boucle la plus interne.

Dans ces algorithmes, $izin$ et $izout$ sont des itérateurs STL, qui permettent de parcourir les conteneurs $infozin$ et $infozout$. Initialiser l'itérateur consiste à le faire pointeur sur le premier élément du conteneur, et l'incrémenter revient à le déplacer sur l'élément suivant.

Les z_{in} et z_{out} sont stockés sous forme de couples (position, valeur), comme cela a été expliqué pour l'algorithme PR-Sh lorsque π' est un vecteur intermédiaire (algorithme 4.6).

III.3. COMPLEXITÉ

Il faut déjà remarquer que cet algorithme n'utilise plus aucun tableau de taille $|\hat{S}|$, comme c'était le cas pour E-Sh et PR-Sh. Tous les tableaux sont de taille $|S|$ si on suppose qu'il n'y a pas de vecteurs intermédiaires $\hat{\pi}$ tels que $|pos_{\hat{\pi}}| > |S|$. Ceci n'arrive que très rarement, et dans ces cas l'algorithme FR-Sh devient très coûteux (réallocation mémoire), donc on se limite à l'étude de la complexité dans les autres cas.

Au niveau des temps d'exécution, on réduit le nombre de multiplications à un ordre de $|S| \times \sum_{i=1}^{N} \frac{nz_i}{n_i}$ lorsque π' est un vecteur de probabilité (on suppose ici que le nombre d'éléments non nuls par colonne est uniforme). Cependant, on introduit des coûts supplémentaires, et en particulier le coût d'un tri qui peut atteindre un ordre de $O(|S|log(|S|))$.

Lorsque le pourcentage d'états non accessibles est élevé ($|S| \ll |\hat{S}|$), l'amélioration est significative. En revanche, le nombre de calculs et le coût mémoire sont plus élevés que pour l'algorithme E-Sh lorsque $|S| \approx |\hat{S}|$.

Nous avons néanmoins rempli notre objectif, consistant à n'utiliser plus aucune structure de taille $|\hat{S}|$, tout en conservant un algorithme efficace lorsque $|S| \ll |\hat{S}|$.

Algorithme 4.7

```
1  j = 0 ; j2 = 0 ;
2  initialize   l_1 = 1, l_2 = 1, ..., l_{i-1} = 1 ;
3  for l = 0, 1, ..., nleft_i - 1          // boucle sur les portions
4  do  effacer infozin et infozout ;
5      initialiser izin et izout ;
       // Remplir infozin
6      while   (j < |S|) && (π.positions[j] < (l + 1) × njump_i)
7      do  re = (π.positions[j] − l × njump_i) mod nright_i ;
8          qo = (π.positions[j] − l × njump_i) div nright_i ;
9          rajouter (re,qo,j) à infozin ;
10         j = j + 1 ;
11     end do
       // Remplir infozout
12     while   (j2 < |S|) && (π'.positions[j2] < (l + 1) × njump_i)
13     do  re = (π'.positions[j2] − l × njump_i) mod nright_i ;
14         qo = (π'.positions[j2] − l × njump_i) div nright_i ;
15         rajouter (re,qo,j2) à infozout ;
16         j2 = j2 + 1 ;
17     end do
18     trier infozin et infozout selon le numéro ;
19     initialize   r_{i+1} = 1, r_{i+2} = 1, ..., r_N = 1 ;
20     for r = 0, 1, ..., nright_i - 1          // détail de la portion : boucle sur les z_in
21     do  (num_o, place_o, index_o) = izout ;
22         if   (num_o == r)
23             effacer le z_in ;
24             (num_i, place_i, index_i) = izin ;
25             while   (num_i == r)      // extraction de z_in
26             do  rajouter (place_i, π.vec[index_i]) à z_in
27                 incrémenter izin ;
28                 (num_i, place_i, index_i) = izin ;
29             end do
30             evaluate   Q^(i)(a_{l_1}^(1), ..., a_{r_N}^(N)) ;
31             while   (num_o == r)      // Calcul du résultat
32             do  π'.vec[index_o] = z_in × Q^(i)[place_o] ; // Multiplication par une colonne
33                 incrémenter izout ;
34                 (num_o, place_o, index_o) = izout ;
35             end do
36         end if
37         next   r_{i+1}, r_{i+2}, ..., r_N ;
38     end do
39     next   l_1, l_2, ..., l_{i-1} ;
40 end do
```

Algorithme 4.7: **FR-Sh** avec fonction - π' est un vecteur de probabilité

Algorithme 4.8

```
1  j = 0 ; j2 = 0 ;
2  initialize   l_1 = 1, l_2 = 1, ..., l_{i-1} = 1 ;
3  for l = 0, 1, ..., nleft_i - 1          // boucle sur les portions
4  do  effacer infozin et infozout ;
5      initialiser izin et izout ;
       // Remplir infozin
6      while   (j < |S|) && (π.positions[j] < (l + 1) × njump_i)
7      do  re = (π.positions[j] − l × njump_i) mod nright_i ;
8          qo = (π.positions[j] − l × njump_i) div nright_i ;
9          rajouter (re,qo,j) à infozin ;
10         j = j + 1 ;
11     end do
12     trier infozin selon le numéro ;
13     initialize   r_{i+1} = 1, r_{i+2} = 1, ..., r_N = 1 ;
14     for r = 0, 1, ..., nright_i - 1          // détail de la portion : boucle sur les z_in
15     do  (num, place, index) = izin ;
16         if   (num == r)
17             effacer le z_in ;
18             while   (num == r)      // extraction de z_in
19             do  rajouter (place, π.vec[index]) à z_in
20                 incrémenter izin ;
21                 (num, place, index) = izin ;
22             end do
23             evaluate   Q^(i)(a_{l_1}^(1), ..., a_{r_N}^(N)) ;
24             multiply   z_out = z_in × Q^(i) ;
25         end if
26         next   r_{i+1}, r_{i+2}, ..., r_N ;
27     end do
28     classer z_out par ordre de positions croissantes ;
29     while   ((pos, val) ∈ z_out)          // Stockage du résultat de la portion
30     do  π'.positions[j2] = pos + l × njump_i ;
31         π'.vec[j2] = val ;
32         j2 = j2 + 1 ;
33     end do
34     if   (j2 ≥ dimension)
35         Prévenir l'utilisateur du débordement ou réallocation dynamique.
36     end if
37     next   l_1, l_2, ..., l_{i-1} ;
38 end do
   // Si nécessaire, met un marqueur pour savoir où s'arrête le tableau utilisé
39 if   (j2 < dimension)
40     π'.positions[j2] = −1
41 end if
```

Algorithme 4.8: **FR-Sh** avec fonction - π' est un vecteur intermédiaire

Maintenant que les algorithmes du shuffle exploitant la structure creuse des vecteurs de probabilité ont été présentés, nous exposons une technique de réordonnancement des automates qui permet d'optimiser l'algorithme du shuffle dans certains cas, en réduisant le nombre d'évaluations de fonction. Cette optimisation sera présentée pour les trois algorithmes **E-Sh**, **PR-Sh** et **FR-Sh**.

IV. OPTIMISATION ALGORITHMIQUE : RÉORDONNANCEMENT DES AUTOMATES

Une des opérations les plus coûteuses effectuées lors d'une multiplication vecteur-descripteur est l'évaluation des fonctions. Dans les algorithmes présentés précédemment, ces évaluations ont lieu dans la boucle la plus interne de l'algorithme, par exemple à la ligne 12 de l'algorithme 4.3.

Des techniques de réordonnancement des automates, présentées dans [41, 42], permettent d'effectuer dans certains cas les évaluations en dehors de la boucle la plus interne.

En observant des cas particuliers de produits tensoriels généralisés, on comprend aisément l'intérêt de cette optimisation. Prenons par exemple le produit tensoriel généralisé :

$$\bigotimes_{i=1}^{N}{}_{g} Q^{(i)}(\mathcal{A}^{(i+1)}, \ldots, \mathcal{A}^{(N)})$$

Les facteurs normaux de la décomposition doivent être traités dans l'ordre suivant :

$$\begin{aligned} & I_{nleft_1} \underset{g}{\otimes} Q^{(1)}(\mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(N)}) \underset{g}{\otimes} I_{nright_1} \\ \times & I_{nleft_2} \underset{g}{\otimes} Q^{(2)}(\mathcal{A}^{(3)}, \ldots, \mathcal{A}^{(N)}) \underset{g}{\otimes} I_{nright_2} \\ \times & \cdots \\ \times & I_{nleft_N} \underset{g}{\otimes} Q^{(N)} \underset{g}{\otimes} I_{nright_N} \end{aligned}$$

De plus, on peut observer que chaque matrice $Q^{(i)}(\ldots)$ ne dépend que des automates à sa droite. Cela nous permet d'éliminer les lignes correspondant au calcul des états locaux des automates à sa gauche (lignes 3 et 22) dans l'algorithme 4.3.

De façon analogue, pour le produit tensoriel généralisé :

$$\bigotimes_{i=1}^{N}{}_{g} Q^{(i)}(\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(i-1)})$$

Les facteurs normaux de la décomposition doivent être traités dans l'ordre suivant :

$$
\begin{aligned}
& I_{nleft_N} \underset{g}{\otimes} Q^{(N)}(\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N-1)}) \underset{g}{\otimes} I_{nright_N} \\
\times & I_{nleft_{N-1}} \underset{g}{\otimes} Q^{(N-1)}(\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N-2)}) \underset{g}{\otimes} I_{nright_{N-1}} \\
\times & \cdots \\
\times & I_{nleft_1} \underset{g}{\otimes} Q^{(1)} \underset{g}{\otimes} I_{nright_1}
\end{aligned}
$$

Puisque les matrices $Q^{(i)}(\ldots)$ ne dépendent que des automates à sa gauche, ce sont les lignes correspondant au calcul des états locaux des automates à sa droite (lignes 5 et 19) qui peuvent être supprimées. De plus, l'évaluation des matrices (ligne 12 de l'algorithme 4.3) peut être déplacée à l'extérieur de la boucle sur r de 0 jusqu'à $nright_i - 1$.

On cherche donc à réordonnancer les automates pour se retrouver dans de tels cas, et diminuer ainsi le nombre d'évaluations de fonction. En effet, un automate ne dépend pas nécessairement de tous les autres automates, il peut ne dépendre que de l'état de certains des automates.

Nous ne présenterons pas ici les techniques de réordonnancement, déjà largement développées dans [41, 42] ; on suppose par la suite que l'on dispose d'un réordonnancement des automates nous permettant de minimiser le nombre d'évaluations de fonction.

Nous exposerons cependant le principe de la permutation d'un vecteur, qu'il s'agisse d'un vecteur étendu ou d'un vecteur réduit. Cette permutation est utilisée par la suite pour expliquer les nouvelles versions de l'algorithme du shuffle avec permutation.

IV.1. PERMUTATION DE VECTEURS

Lorsque les automates changent d'ordre, les éléments du vecteur doivent changer de place car l'ordre lexicographique est modifié. Nous commençons par détailler la notion d'ordre lexicographique, puis nous donnons l'algorithme de permutation de vecteurs.

IV.1.1. ORDRE LEXICOGRAPHIQUE

Le tableau ci-dessous montre les états d'un vecteur pour les automates ordonnés de deux façons différentes. Les éléments du vecteur sont triés suivant un ordre lexicographique exprimé par la liste des automates dans un certain ordre. A gauche, on a utilisé l'ordre $\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \mathcal{A}^{(3)}$. Les automates sont de taille 2, 3, 2 respectivement. A droite, on a effectué un réordonnancement. Les automates sont dans l'ordre $\mathcal{A}^{(3)}, \mathcal{A}^{(1)}, \mathcal{A}^{(2)}$.

$\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \mathcal{A}^{(3)}$				$\mathcal{A}^{(3)}, \mathcal{A}^{(1)}, \mathcal{A}^{(2)}$			
rang	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	rang	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
0	0	0	0	0	0	0	0
1	0	0	1	1	0	1	0
2	0	1	0	2	0	2	0
3	0	1	1	3	1	0	0
4	0	2	0	4	1	1	0
5	0	2	1	5	1	2	0
6	1	0	0	6	0	0	1
7	1	0	1	7	0	1	1
8	1	1	0	8	0	2	1
9	1	1	1	9	1	0	1
10	1	2	0	10	1	1	1
11	1	2	1	11	1	2	1

Quel que soit l'ordre utilisé, il est toujours facile d'incrémenter de +1 l'état du vecteur. En effet, un incrément du vecteur correspond à un incrément d'un état local, avec éventuellement propagation de retenue.

Pour effectuer l'incrément, on regarde les automates du dernier au premier, et on essaie d'incrémenter l'état local de l'automate. Un incrément est invalide quand l'état local de l'automate est le dernier état. Lorsque l'incrément est invalide, l'état local de l'automate est remis au premier état (*reset* de l'état), et on considère l'automate suivant (donc l'automate qui précède suivant l'ordre des automates). La fonction s'arrête dès qu'un incrément valide peut avoir lieu. Dans l'exemple qui précède, et pour l'ordre $\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \mathcal{A}^{(3)}$, on cherche par exemple à incrémenter de +1 l'état $(0, 1, 1)$. L'automate $\mathcal{A}^{(3)}$ est le premier que l'on essaie d'incrémenter. Mais cet automate est dans son dernier état, on met donc sa valeur à 0. On essaie ensuite d'incrémenter $\mathcal{A}^{(2)}$, et l'incrément est valide. L'état obtenu est donc l'état $(0, 2, 0)$.

Lorsqu'on effectue pour une permutation un incrément suivant un ordre lexicographique, on désire calculer le rang de l'état obtenu dans le vecteur permuté, donc suivant un autre ordre lexicographique. On calcule ce nouveau rang à partir du rang courant dans le vecteur permuté. Dans notre exemple, le rang de l'état $(0, 1, 1)$ par le nouvel ordre lexicographique $\mathcal{A}^{(3)}, \mathcal{A}^{(1)}, \mathcal{A}^{(2)}$ est 7. Le rang de l'état après incrément (état $(0, 2, 0)$) est 2. Pour calculer le nouveau rang, on procède par incréments et décréments correspondant aux incréments et *reset* effectués sur les états locaux des automates. Les valeurs de ces incréments et décréments dépendent du nouvel ordre lexicographique. Un incrément valide de +1 sur un état local de l'automate $\mathcal{A}^{(i)}$ correspond à une addition de $nright_i$ sur le rang, et un *reset* de l'état de l'automate $\mathcal{A}^{(i)}$ correspond à une soustraction de $nright_i \times n_i$ sur le rang. Dans notre exemple, on fait un *reset* de l'état de l'automate $\mathcal{A}^{(3)}$, et dans le nouvel ordre lexicographique, $nright_3 = 2 \times 3 = 6$. On obtient donc l'état de rang $7 - 6 = 1$. On incrémente ensuite la valeur de l'état de l'automate $\mathcal{A}^{(2)}$, pour lequel $nright_2 = 1$ dans le nouvel ordre lexicographique. On incrémente donc le rang de 1, on obtient 2.

On désire effectuer la permutation du vecteur en parcourant ce vecteur séquentiellement. On connaît le rang de l'état initial suivant le nouvel ordre lexicographique. On ef-

fectue ensuite un incrément de +1 sur l'état du vecteur à permuter, et on calcule le rang de l'état suivant d'après le nouvel ordre des automates. L'algorithme de permutation de vecteurs étendus est triviale, il est présenté dans le paragraphe qui suit. Nous verrons ensuite comment nous avons adapté cet algorithme au cas des vecteurs réduits.

IV.1.2. PERMUTATION D'UN VECTEUR ÉTENDU

Pour effectuer la permutation d'un vecteur étendu (de la taille de $\hat{S}$), on exploite le changement d'ordre lexicographique. L'algorithme crée une nouvelle structure de vecteur (un tableau de taille $\hat{S}$). On parcourt alors séquentiellement le vecteur à permuter $\hat{\pi}$, et on regarde quelle est la nouvelle place de chaque élément dans le vecteur permuté $\hat{\pi}'$ (rang idx). Pour connaître cette place, on incrémente de +1 l'état courant ($state$), et on calcule le rang (idx) de cet état dans le vecteur permuté.

La figure 4.11 et l'algorithme 4.9 présentent la permutation d'un vecteur étendu.

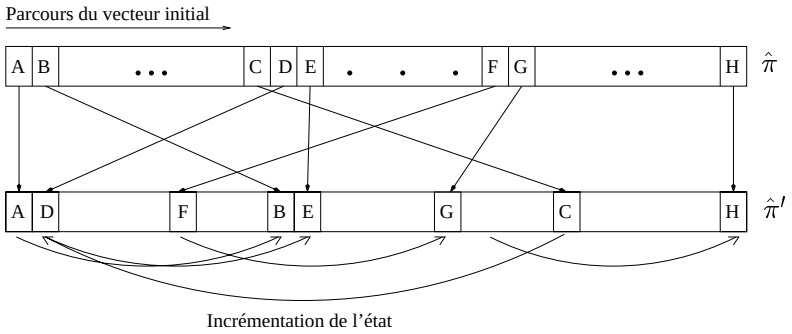


FIG. 4.11 – Permutation d'un vecteur étendu $\hat{\pi}$

Algorithme 4.9

```
1  créer un nouveau tableau π̂' de taille |Ŝ|
2  initialiser state ;          // état initial dans π̂
3  idx = 0 ;                    // indice dans π̂' (rang de state dans le vecteur permuté)
4  for i = 0, 1, ..., |Ŝ|       // parcours séquentiel de π̂
5  do  π̂'[idx] = π̂[i] ; // élément du vecteur à la place i : maintenant à la place idx
6      incrémenter l'état state de +1
7      mettre à jour le rang idx suivant le nouvel ordre lexicographique
8  end do
9  π̂ = π̂' ;
```

Algorithme 4.9: Permutation d'un vecteur étendu $\hat{\pi}$

IV.1.3. PERMUTATION D'UN VECTEUR RÉDUIT

Dans l'algorithme 4.9, on effectue des accès au vecteur $\hat{\pi}'$ pour stocker l'élément correspondant au rang idx dans ce vecteur (ligne 5). Lorsque le vecteur est réduit, cette opération n'est plus possible car on ne stocke plus que les états accessibles, et on n'a pas de correspondance directe entre idx et l'indice de l'état dans le vecteur réduit. Ce dernier indice dépend en effet du nombre d'états accessibles classés avant l'état de rang idx suivant le nouvel ordre lexicographique dans le vecteur résultat.

On choisit cependant de fonctionner de la même façon que l'algorithme de permutation de vecteur étendu. On effectue ainsi une boucle sur $\hat{S}$. On procède par incréments de +1 de l'état du vecteur ($state$). On peut, comme précédemment, calculer le rang de cet état dans le vecteur permuté (idx). Lorsque l'état du vecteur π est accessible (valeur présente dans le vecteur π qui est stocké en creux), on stocke le couple (nouvelle position, valeur) dans un vecteur STL appelé π'. Une fois cette boucle terminée, on dispose de toutes les valeurs à stocker dans le vecteur résultat, ainsi que leurs positions. Cependant, ces positions ne sont pas forcément dans un ordre croissant. On effectue donc un tri de ce vecteur par positions croissantes, puis on stocke le résultat dans le vecteur réduit π (tableaux $\pi.positions$ et $\pi.vec$). L'itérateur $iter$ permet de parcourir le vecteur STL π'.

L'algorithme 4.10 effectue cette permutation.

IV.2. ALGORITHMES DE MULTIPLICATION VECTEUR-DESCRIPTEUR

Une fois que l'algorithme de permutation d'un vecteur a été implanté, on peut appliquer la technique de réordonnancement, et donc effectuer les évaluations de fonction en dehors de la boucle la plus interne. On ne fait plus qu'une évaluation de matrice par portion.

Les modifications à apporter aux algorithmes présentés dans les sections précédentes sont maintenant évoquées.

IV.2.1. ALGORITHME E-Sh (SECTION I)

Dans le cas de l'algorithme E-Sh, il suffit de changer de place la ligne correspondant à l'évaluation de la matrice dans l'algorithme 4.3, pour la sortir de la boucle la plus interne. La ligne 12 est donc déplacée entre les lignes 5 et 6.

IV.2.2. ALGORITHME PR-Sh (SECTION II)

Dans le cas où l'on obtient un vecteur intermédiaire, l'algorithme **PR-Sh** (algorithme 4.6) est modifié par un simple déplacement de la ligne 13 entre les lignes 11 et 12 (on sort l'évaluation de la matrice de la boucle la plus interne).

En revanche, une simple modification de l'algorithme 4.5 n'est pas la solution la plus

Algorithme 4.10

```
1   créer un vecteur STL π' (il sera de taille |S|)
2   initialiser state ;          // état initial dans π
3   idx = 0 ;          // indice dans π' (rang de state dans le vecteur permuté)
4   k = 0 ;          // pointeur sur l'élément courant du vecteur réduit
5   for i = 0, 1, ..., |Ŝ|
6   do  if   (i == positions[k])          // on a atteint un état accessible
7           rajouter (idx, vec[k]) à π'
8           k = k + 1 ;
9       end if
10      incrémenter l'état state de +1
11      mettre à jour le rang idx suivant le nouvel ordre lexicographique
12  end do
13  trier π' et initialiser iter au début de π'
14  k = 0 ;
    // on stocke les éléments de π' dans le vecteur réduit π
15  while   (iter n'est pas à la fin de π')
16  do  (pos, val) = iter ;
17      π.positions[k] = pos ;
18      π.vec[k] = val ;
19      k = k + 1 ; incrémenter iter ;
20  end do
```

Algorithme 4.10: Permutation d'un vecteur réduit π

appropriée dans le cas où l'on obtient un vecteur de probabilité. En effet, l'algorithme 4.4 sans évaluation de fonction est beaucoup plus simple que l'algorithme 4.5, mais nous avons du écrire ce dernier pour pouvoir recréer la boucle interne dans laquelle les évaluations de fonction devaient être faites. Nous étions alors obligés de traiter les z_{in} dans un ordre séquentiel, et surtout de calculer tous les éléments résultats obtenus par la multiplication d'un même z_{in} par une colonne de la matrice les uns à la suite des autres, pour éviter des réévaluations coûteuses.

Lorsqu'on ne fait qu'une seule évaluation de matrice par portion, l'ordre dans lequel on traite les z_{in} n'a plus d'importance. On peut donc reprendre l'algorithme 4.4, qui effectue le calcul des éléments du résultat séquentiellement, sans se soucier de l'ordre des z_{in}. Il suffit alors de rajouter à cet algorithme les initialisations des vecteurs, les incréments des sous vecteurs gauche et droite, et l'évaluation de la matrice entre les lignes 7 et 8.

IV.2.3. ALGORITHME FR-Sh (SECTION III)

La structure de l'algorithme peut être conservée, on se contente ici de sortir l'évaluation de la boucle la plus interne en déplaçant la ligne correspondante.

Ainsi, dans le cas où l'on obtient un vecteur de probabilité (algorithme 4.7), on déplace la ligne 30 entre les lignes 19 et 20. Lorsqu'on obtient un vecteur intermédiaire (algorithme 4.8), on déplace la ligne 23 entre les lignes 14 et 15. Ainsi, on n'effectue plus qu'une seule évaluation de matrice par portion.

V. CONCLUSION

Nous avons ainsi présenté dans ce chapitre l'algorithme du shuffle standard E-Sh, ainsi que deux variantes exploitant le fait qu'un grand nombre d'états peuvent ne pas être accessibles. Ces nouvelles versions de l'algorithme du shuffle travaillent sur des vecteurs *réduits*, vecteurs de la taille de l'espace d'états accessibles S.

L'algorithme PR-Sh cherche uniquement à améliorer la complexité en temps d'exécution lorsque cela est possible, alors que FR-Sh propose une optimisation de la place mémoire utilisée. Nous avons également présenté un algorithme de permutation de vecteurs réduits, qui permet d'adapter des techniques de réordonnancement d'automates aux algorithmes utilisant des vecteurs réduits.

Le chapitre suivant effectue toute une série de tests de ces algorithmes sur différents exemples, pour tester leurs avantages respectifs. Des comparatifs avec d'autres algorithmes de multiplication vecteur-descripteur sont également effectués.

CHAPITRE

Résultats numériques

5

Dans cette section, nous présentons quelques résultats numériques obtenus avec les nouvelles méthodes et algorithmes que nous avons présentés dans cette thèse pour évaluer les performances de systèmes à grand espace d'état :

– l'agrégation de SANs avec réplication (chapitre 3) [4] ;
– l'algorithme du shuffle utilisant des *vecteurs réduits* (chapitre 4) [7, 8].

Pour cela, nous commençons par décrire le logiciel PEPS (*Performance Evaluation of Parallel Systems*). Nous avons rajouté à ce logiciel les nouvelles méthodes et algorithmes afin de pouvoir obtenir des résultats numériques.

Des logiciels similaires existent pour traiter des modèles sous d'autres formalismes, et notamment les réseaux de Pétri [26, 29, 3, 57].

La section II compare les différents algorithmes de multiplication vecteur-descripteur, puis la section III donne quelques résultats numériques pour montrer l'intérêt de l'agrégation des SANs avec réplication.

I. PEPS 2003

PEPS est un outil informatique qui permet à la fois la définition et la solution de modèles utilisant le formalisme des réseaux d'automates stochastiques (chapitre 2). Le formalisme SAN définit une façon compacte de stocker la matrice de transition de la chaîne de Markov dans le cas du temps continu, et utilise ainsi l'algèbre tensorielle pour effectuer les multiplications vecteur matrice.

La première version du logiciel est proposée en 1988 [46], puis le logiciel connaît de grandes modifications avec la version PEPS 2.0 [41], puis PEPS 2000.

Nous présentons ici la version PEPS 2003 [5], qui possède des nouvelles fonctionnalités par rapport à PEPS 2000. L'interface a été revue pour permettre facilement la définition de SANs avec réplication (section I.1), et les nouveaux algorithmes du shuffle ont été implémentés. Nous présentons dans la section I.2 les principales caractéristiques de PEPS 2003.

> *Cette présentation se limite au cadre du temps continu ; l'interface est cependant facilement adaptable pour décrire les modèles à temps discret. En revanche, les méthodes de résolution pour le temps discret n'ont pas encore été développées.*

I.1. INTERFACE

Le logiciel propose un formalisme textuel pour décrire les modèles, qui conserve l'élément clé du formalisme SAN, à savoir sa spécification modulaire. La description textuelle est simple, extensible et flexible :

- simple parce qu'il y a peu de mots réservés, juste assez pour délimiter les différents niveaux de modularité ;
- extensible car la définition des modèles SAN est faite de façon hiérarchique ;
- flexible grâce à l'inclusion de structures de réplication, qui permettent d'une part la réutilisation d'automates identiques, et d'autre part la construction d'automates ayant des blocs d'états de comportement identique répétés, comme on le trouve souvent dans les modèles de réseaux de file d'attente.

Nous commençons par présenter la syntaxe du format textuel, puis un exemple illustre cela.

I.1.1. FORMAT TEXTUEL

La figure 5.1 présente la structure modulaire du format textuel de PEPS 2003. La description d'un SAN est constituée de cinq blocs, délimités par les mots en **gras**. Les autres mots réservés du langage PEPS sont en *italique*. Les mots entre "<" et ">" sont des informations qui doivent être fournies par l'utilisateur, et les symboles "{" et "}" indiquent des informations facultatives.

Le premier bloc **identifiers** contient les déclarations de tous les paramètres : valeurs numériques, fonctions, ou ensemble d'indices (domaines) qui seront utilisés pour les répliquas dans la définition du modèle. Un identifiant ($< id\_name >$) peut être n'importe quelle chaîne de caractères. Les valeurs numériques et les fonctions sont définies selon une syntaxe identique au langage C. Les expressions sont similaires à des expressions mathématiques communes, avec des opérateurs logiques et arithmétiques. Les arguments de ces expressions peuvent être des nombres constants (paramètres du modèle), mais aussi des automates ou des identifiants d'état. Dans ce dernier cas, les expressions sont des fonctions définies sur l'espace d'état du SAN. Par exemple, "le nombre d'automates dans l'état x" (qui donne un résultat entier) s'exprime par "$nb\ x$". La fonction "$(st\ A1! = st\ A2) * 4$" retourne la valeur 4 si les deux automates $A1$ et $A2$ sont dans des états différents, et 0 sinon (les opérateurs de comparaison retournent 1 pour un résultat vrai, et 0 sinon).

Les ensemble d'indices sont utiles pour définir des événements, automates ou états décrits avec des réplications. Ainsi, un groupe de répliquas de l'automate A avec l'ensemble d'indices $[0..2, 5, 8..10]$ définit les sept automates suivant : $A[0]$, $A[1]$, $A[2]$, $A[5]$, $A[8]$, $A[9]$ et $A[10]$.

identifiers
$< id\_name >=< exp >$ **;**
$< id\_name >= [domain]$ **;**

events
// sans réplication
loc $< evt\_name >< rate >< automaton >$
syn $< evt\_name >< rate >< automata >$
// avec réplication
loc $< evt\_name > [domain] < rate >< automata > [domain]$
syn $< evt\_name > [domain] < rate >< automata > [exp - domain]$

{*partial*} **reachability** $=< exp >$ **;**

network $< net\_name > (< type >)$
$aut < aut\_name > \{[domain]\}$
stt $< stt\_name > \{[domain]\}$
to($< stt\_name >$ *)*
$< evt\_name > \{< prob >\}$
from $< stt\_name >$
to($< stt\_name >$ *)*
$< evt\_name > \{< prob >\}$

results
$< res\_name >=< exp >$ **;**

FIG. 5.1 – Structure modulaire du format textuel de PEPS 2003

Le bloc **events** définit chaque événement du modèle, précisant pour un événement e :
– son type : événement local (*loc*) ou synchronisant (*syn*) ;
– son nom (e) ;
– son taux de franchissement (λ_e) ;
– l'ensemble d'automates impliqués par cet événement O_e.

Nous pouvons également répliquer les événements en utilisant des ensembles d'indices. Ceci est notamment utilisé lorsqu'un ensemble d'automates possèdent des événements avec un même taux de franchissement.

Le bloc **reachability** définit la fonction d'accessibilité du SAN. C'est une fonction qui renvoie une valeur non nulle pour les états de S, et la valeur 0 pour les états de$\hat{S} - S$. Nous pouvons définir une fonction d'accessibilité partielle en rajoutant le mot-clé "partial". Dans ce cas, seul un sous-ensemble de S est défini, et l'ensemble des états accessibles est ensuite généré par le logiciel.

Le bloc **network** est la partie principale du SAN, il a une structure hiérarchique. Un réseau d'automates stochastiques est constitué d'un ensemble d'automates ; chaque automate est constitué d'un ensemble d'états ; chaque état a un ensemble d'arcs sortant (transitions) ; et chaque arc est étiqueté par un ensemble d'événements. La première ligne contient des in-

formations générales sur le SAN : le nom du modèle "$< net\_name >$" et le type d'échelle de temps du modèle, qui peut être "*continuous*" ou "*discrete*" suivant que le modèle soit à temps continu ou à temps discret. Pour l'instant, seule l'analyse des modèles à temps continu est implémentée dans PEPS 2003.

Le mot-clé *aut* sert à délimiter les automates. "$< aut\_name >$" est le nom de l'automate, et nous pouvons éventuellement donner un ensemble d'indices $[domain]$ pour répliquer l'automate, *i.e.*, pour créer un certain nombre de copies de l'automate. Dans ce cas, si i est un indice de $[domain]$ et A le nom de l'automate, alors $A[i]$ est l'identifiant de l'un des automates répliqués.

Le mot-clé *stt* annonce la définition d'un état. "$< stt\_name >$" est le nom de l'état, et nous pouvons utiliser $[domain]$ pour répliquer l'état.

Une description de chaque arc sortant de cet état est donnée par la définition d'une section "*to()*". L'identifiant "$< stt\_name >$" dans la parenthèse indique l'état d'arrivée de l'arc. Dans un groupe d'états répliqués, l'expression d'autres états du groupe peut être faite par des références aux positions : état courant (==), précédent (– –) ou suivant (++). Des sauts plus importants, par exemple un saut de deux états peuvent également être définis (+2), mais il faut bien noter que toute référence à une position qui pointe sur un état non existant ou un état en dehors du groupe d'états répliqués est ignorée.

Nous associons alors à chaque arc un ensemble d'événements qui peuvent déclencher la transition. Ils sont exprimés par leur nom ("$< evt\_name >$"), et éventuellement (si différente de 1) par la probabilité de routage associée à cette transition ($< prob >$). Les différents événements sont séparés par des espaces ou des sauts de ligne.

La section *from* est relativement semblable à la section *stt*, sauf qu'elle ne définit pas d'état locaux. Elle sert à rajouter des arcs qui ne peuvent pas être définis dans une section *stt*. Typiquement, nous nous en servons pour définir un arc sortant d'un état particulier d'un groupe d'états répliqués pour aller dans un état en dehors du groupe. Une file d'attente ayant des états initiaux ou finaux particuliers peut utiliser ce type de définition d'arcs.

Enfin, les fonctions utilisées pour calculer des indices de performance sur le SAN sont définies dans le bloc **results**. Les résultats calculés par PEPS 2003 sont les valeurs moyennes de ces fonctions, calculées en utilisant le vecteur des probabilités stationnaires du modèle.

I.1.2. EXEMPLE

Avec l'interface de PEPS 2003, le format textuel qui décrit le SAN de la figure 2.4 page 33 est :

```
identifiers
   lambda1 = 3;
   lambda2 = 4;
   lambda3 = 6;
   lambda4 = 5;
   mu1     = 2;
   mu2     = 7;
   pi      = 0.4;
```

```
   f       = ((st A1 == s0) * mu1) + ((st A1 == s2) * mu2);


events
   loc loc1 (lambda1)  A1
   loc loc2 (lambda2)  A1
   loc loc3 (lambda3)  A1
   syn evts (lambda4)  A1 A2
   loc evtf (f)        A2

reachability = 1;

network study_exemple (continuous)
  aut A1 stt s0 to(s1) loc1
         stt s1 to(s2) loc2
         stt s2 to(s0) loc3 evts(1-pi)
                to(s1)      evts(pi)
  aut A2 stt t0 to(t1) evtf
         stt t1 to(t0) evts

results
  A1_in_so_and_A2_in_t0 = (st A1 == s0) && (st A2 == t0);
```

I.2. LE LOGICIEL PEPS 2003

PEPS est implémenté en langage C++ [91], et bien que le code source soit relativement standard, seules les versions Linux et Solaris ont été testées. Les principales fonctions de PEPS 2003 sont :

- la lecture d'une **description textuelle** compacte des SANs à temps continu, comportant des structures de réplication (présentée en section I.1) ;
- la **compilation** du modèle SAN pour former toutes les matrices du descripteur du SAN ;
- une **multiplication vecteur-descripteur** efficace, basée sur l'algorithme du shuffle, qui utilise au choix des vecteurs *étendus* (de la taille de $\hat{S}$) ou *réduits* (de la taille de l'espace d'états accessibles S), cf section 4 ;
- différentes **méthodes de résolution itératives** : méthode de la puissance, Arnoldi, et GMRES [84, 89, 19] ;
- des **optimisations numériques** tenant compte des dépendances fonctionnelles, un pré-calcul de la diagonale, des techniques de préconditionnement, une agrégation algébrique des automates [41, 42, 43, 24], et une évaluation rapide des fonctions [5] ;
- la **génération** de la matrice générateur du SAN en format HBF ou dans un format compatible avec le logiciel MARCA [90] ;
- l'évaluation des **résultats** à partir du vecteur des probabilités stationnaires.

II. MULTIPLICATION VECTEUR-DESCRIPTEUR

Nous effectuons dans cette section une étude des performances des différents algorithmes du chapitre 4 pour pouvoir les comparer entre eux. Nous comparons à la fois les besoins en mémoire et les durées d'exécution des différents algorithmes. Les résultats obtenus en générant la matrice globale en format Harwell-Boeing (HBF) puis en effectuant une multiplication classique d'un vecteur par une matrice creuse [84] sont également exposés à titre comparatif.

Nous présentons dans un premier temps les conditions dans lesquelles les tests ont été réalisés. Ensuite, nous exposons les résultats des tests à travers les modèles classiques extraits de la littérature ([41, 42, 6]) et présentés dans la section IV du chapitre 2 (page 53). Seuls les modèles à temps continu sont étudiés étant donné que les méthodes développées ne concernent que cette classe de modèles.

II.1. CONDITIONS DES MESURES

Nous avons réalisé toutes les mesures numériques à l'aide du logiciel PEPS 2003 ([46, 5]), dans lequel nous avons implémenté les nouvelles versions de l'algorithme du shuffle, la génération de la matrice en format HBF à partir du descripteur, et la multiplication d'un vecteur par une matrice en format HBF. Nous n'avons pas employé l'évaluation rapide des fonctions, développée ultérieurement. Tous les temps d'exécution ont été mesurés avec une précision du dixième de seconde sur un PC Pentium III à 531 MHz, sous un système Linux Mandrake 2.2.14, et avec 128 MegaOctets de mémoire vive. La convergence a été vérifiée avec une précision absolue à la dixième décimale, i.e., les résultats ont une tolérance de l'ordre de 10^{-10}. Les vecteurs initiaux de toutes les expériences sont des vecteurs équiprobables.

Les changements que nous avons implantés dans PEPS ne concernent que la multiplication vecteur-descripteur. Par conséquent, les facteurs de gain ou perte en performances des nouveaux algorithmes sont identiques pour toutes les méthodes itératives (Puissance, Arnoldi, GMRES) et pour tous les types de préconditionnement. Ainsi, nous nous sommes limités aux mesures à l'aide d'une seule méthode itérative : *la méthode de la puissance sans préconditionnement*.

II.2. PRÉSENTATION DES RÉSULTATS

Les résultats sont présentés dans deux tableaux. Le premier expose les résultats en terme de temps d'exécution, et le deuxième présente les performances au niveau mémoire. Les

deux tableaux se présentent sous la même forme :

- La première colonne explicite les paramètres du modèle, ainsi que le nombre d'états du système (colonne $|\hat{S}|$), le nombre d'états accessibles (colonne $|S|$), la proportion d'états accessibles du modèle (colonne $\% = \frac{|S|}{|\hat{S}|} \times 100$).
- La colonne E-Sh (*extended* shuffle) présente les résultats sur l'algorithme du shuffle qui utilise des vecteurs étendus. Cet algorithme classique est décrit dans la section I du chapitre 4.
- La colonne PR-Sh (*partially reduced* shuffle) présente les résultats sur les algorithmes de la section II du chapitre 4, qui optimise la durée d'exécution en tenant compte de la proportion d'états accessibles, mais qui utilise des structures intermédiaires en format étendu.
- Les résultats de la colonne FR-Sh (*fully reduced* shuffle) sont obtenus à l'aide de l'algorithme présenté dans la section III du chapitre 4, qui utilise uniquement des structures de données réduites.
- Enfin, la colonne HBF expose les résultats obtenus à l'aide de l'algorithme consistant à générer la matrice en format HBF, puis à effectuer la multiplication standard d'un vecteur par une matrice creuse. Un algorithme de génération HBF est implémenté dans le logiciel PEPS. Il calcule chaque valeur susceptible d'être non nulle à partir du descripteur, en évitant les calculs redondants.

Le premier tableau expose les résultats concernant la durée d'exécution des programmes pour résoudre le modèle. Une première sous-colonne expose le temps de compilation (comp) pour les colonnes E-Sh, PR-Sh et FR-Sh, et le temps de génération (géné) de la matrice HBF pour la colonne HBF. La deuxième sous-colonne présente la durée d'exécution (exec).

Le deuxième tableau présente les résultats sur l'occupation mémoire. Une première sous-colonne présente les résultats sur les structures mémoires stockées sur le disque : la taille du descripteur (des) pour les colonnes E-Sh, PR-Sh et FR-Sh, et la taille de la matrice HBF (mat) pour la colonne HBF. La valeur donnée dans la deuxième sous-colonne (exec) est relevée dans le système pendant que les programmes tournent. Elle représente l'ensemble de la mémoire utilisée par PEPS pendant son exécution (pendant la résolution d'un modèle). Ceci comprend les données (le modèle est stocké en mémoire), les structures mémoire réservées par la procédure, et aussi la pile du processus, ... Les seuls paramètres qui changent d'un algorithme à un autre sont les structures mémoire réservées par les algorithmes (vecteurs, tableaux d'informations intermédiaires, ...).

II.3. RS1-1

Ce premier modèle de partage de ressources est présenté dans la section IV.1.1 du chapitre 2 (page 54) et dans la section I.1.3 du chapitre 3 (page 68).

Ce modèle ne possède pas d'événements synchronisant. Dans les algorithmes du shuffle réduit, les vecteurs obtenus durant les calculs sont donc toujours des vecteurs de probabilité (les vecteurs intermédiaires sont induits par les événements synchronisant). Ce modèle

comporte en revanche des fonctions, il nous permet donc de tester les algorithmes avec fonction. Nous comparons l'emploi des méthodes avec et sans permutation pour les évaluations de fonctions.

Pour les tests, nous utilisons les valeurs $\lambda = 6$ et $\mu = 9$. Le nombre de processus N_p et le nombre de ressources N_r varient suivant les expériences.

II.3.1. MÉTHODE SANS PERMUTATION

Les premiers tests ont été effectués à l'aide de la méthode sans permutation, sur un modèle avec 16 processus, et un nombre de ressources variant entre 1 et 16.

Durée d'exécution (sec) - RS1-1												
Modèle					E-Sh		PR-Sh		FR-Sh		HBF	
N_p	N_r	$\lvert\hat{S}\rvert$	$\lvert S\rvert$	%	comp	exec	comp	exec	comp	exec	géné	exec
16	1	65,536	17	0.03%	2.4	27.2	0.02	1	0.02	3.5	1.7	0
16	4	65,536	2,517	3.8%	2.4	112	0.14	11.5	0.14	22.1	8.5	0.1
16	6	65,536	14,893	22.7%	2.4	178.2	0.68	70	0.68	93.1	34.2	1.5
16	8	65,536	39,203	59.8%	2.4	235.4	1.6	206.6	1.6	267.4	95.3	6.2
16	10	65,536	58,651	89.5%	2.7	292.7	2.3	348.6	2.3	478.9	137.4	13.2
16	12	65,536	64,839	98.9%	2.6	332	2.5	419.6	2.5	612.9	143.5	17
16	16	65,536	65,536	100%	3.7	27	3.7	84.9	3.7	377.8	144.3	19.6

Cet exemple nous expose les performances des algorithmes avec différentes proportions d'états accessibles. Au niveau de la compilation et de la génération de matrices, on constate que la compilation d'un modèle est beaucoup plus rapide que la génération d'une matrice HBF. La compilation est identique pour les algorithmes PR-Sh et FR-Sh, et plus rapide que la compilation pour l'algorithme E-Sh. En effet, l'algorithme E-Sh a besoin de générer tous les éléments de la diagonale, alors que PR-Sh et FR-Sh stockent uniquement les éléments de la diagonale correspondant aux états accessibles, donc ne calculent que ces éléments là lors de la compilation.

La génération de la matrice en format HBF est relativement longue. C'est le principal obstacle à l'application de cette méthode, qui permet d'obtenir de grandes performances au niveau du temps d'exécution. En effet, les évaluations de fonction sont réalisées une fois pour toutes lors de la génération de la matrice, puis on effectue de simples multiplications vecteur/matrice, alors que les trois autres algorithmes effectuent les évaluations de fonction lors de chaque multiplication vecteur/descripteur. Sur ce modèle RS1-1 avec $N_p = 16$, l'algorithme HBF est donc le plus performant au niveau du temps d'exécution.

A propos des algorithmes PR-Sh et FR-Sh qui utilisent des vecteurs réduits, on constate comme prévu que l'algorithme PR-Sh est plus rapide que l'algorithme FR-Sh. Ces algorithmes sont plus rapides que l'algorithme E-Sh tant que le pourcentage d'états accessibles reste raisonnable (inférieur à 60% pour PR-Sh, inférieur à 50% pour FR-Sh). L'intérêt de ces algorithmes ne réside cependant pas dans la résolution de petits modèles comme celui-ci, où HBF est meilleur lorsque le modèle comporte peu d'états accessibles, et où E-Sh est meilleur lorsque le modèle comporte une majorité d'états accessibles.

Occupation mémoire (Ko) - RS1-1												
Modèle					E-Sh		PR-Sh		FR-Sh		HBF	
N_p	N_r	$\|\hat{S}\|$	$\|S\|$	%	des	exec	des	exec	des	exec	mat	exec
16	1	65,536	17	0.03%	522	4,208	10	4,476	10	2,124	0	2,084
16	4	65,536	2,517	3.8%	522	4,208	30	4,584	30	2,304	255	2,380
16	6	65,536	14,893	22.7%	522	4,208	126	5,116	126	3,128	2,086	4,416
16	8	65,536	39,203	59.8%	522	4,208	316	6,176	316	4,748	6,756	9,460
16	10	65,536	58,651	89.5%	522	4,212	468	7,016	468	6,040	11,350	14,356
16	12	65,536	64,839	98.9%	522	4,212	516	7,280	516	6,448	13,085	16,188
16	16	65,536	65,536	100%	522	4,212	522	7,832	522	6,516	13,312	16,436

Au niveau de la mémoire, on constate tout d'abord que la taille du descripteur est fixe dans le cas de l'algorithme E-Sh, alors qu'il dépend du nombre d'états accessibles pour les autres algorithmes. Ceci est lié au fait qu'on ne stocke dans le cas de PR-Sh et FR-Sh que les éléments de la diagonale correspondant aux états accessibles, alors que E-Sh stocke tous les éléments de la diagonale. Pour HBF, la taille de la matrice dépend directement du nombre d'éléments non nuls dans la matrice, et ce nombre dépend du nombre d'états accessibles. On remarque que la place mémoire occupée par la matrice HBF est rapidement élevée, en comparaison de la taille du descripteur.

Pendant l'exécution, on constate que l'algorithme FR-Sh permet une réduction significative de la place mémoire occupée pendant la résolution du modèle par rapport à l'algorithme E-Sh tant que la proportion d'états accessibles est suffisamment faible. En effet, lorsqu'on utilise des vecteurs réduits, les structures en mémoire sont de la taille de S, mais un élément du vecteur nécessite des informations supplémentaires (et notamment sa position dans $\hat{S}$), qui sont elles aussi stockées. Ainsi, lorsqu'on dépasse 50% d'états accessibles, on ne peut pas espérer avoir un gain en mémoire vu qu'un vecteur est stocké à l'aide de deux tableaux de taille $|S|$, ce qui est alors supérieur à $|\hat{S}|$. Les structures intermédiaires utilisées dans PR-Sh rendent son occupation mémoire supérieure dans tous les cas à celle de l'algorithme E-Sh. L'algorithme HBF utilise encore moins de mémoire que FR-Sh lorsqu'il y a très peu d'états accessibles, mais la mémoire utilisée grandit plus vite lorsque le nombre d'états accessibles augmente, pour finalement dépasser la place mémoire occupée par FR-Sh lorsque $N_r > 4$.

Nous avons ensuite effectué des tests sur des modèles plus gros pour cerner la limite et les possibilités des différents algorithmes. Le trait (-) signifie que l'algorithme n'a pas pu aboutir, faute de place mémoire. Pour $N_p = 24$, seules les colonnes FR-Sh et HBF apparaissent dans le tableau, car les deux autres algorithmes n'aboutissent pas.

Durée d'exécution (sec) - RS1-1												
Modèle					E-Sh		PR-Sh		FR-Sh		HBF	
N_p	N_r	$\|\hat{S}\|$	$\|S\|$	%	comp	exec	comp	exec	comp	exec	géné	exec
20	1	1,048,576	21	0.002%	51.8	601.7	0.2	21.2	0.2	64.4	0.6	0
20	4	1,048,576	6,196	0.6%	56.4	2,266.7	0.6	105	0.6	263.6	146	0.2
20	6	1,048,576	60,460	5.8%	59.5	3,285.1	3.9	440.9	3.9	698	1515.4	4.7
20	8	1,048,576	263,950	25.2%	62.9	4,675.1	15.8	1,952.7	15.8	2,604.2	7,552.1	36.8
20	10	1,048,576	616,666	58.8%	58.6	6,154.7	34.1	5,160.1	34.1	7,047.5	-	-
20	20	1,048,576	1,048,576	100%	92.9	586.6	88	2,159.5	88	12,585.3	-	-

Modèle					FR-Sh		HBF	
N_p	N_r	$\|\hat{S}\|$	$\|S\|$	%	comp	exec	géné	exec
24	1	16,777,216	25	0.0001%	2.6	1,100.9	11.8	0
24	4	16,777,216	12,952	0.08%	3.7	3,986.6	4,887.1	0.5
24	6	16,777,216	190,051	1.1%	18.7	7,018.6	73,066.5	12.7
24	8	16,777,216	1,271,626	7.6%	102.3	19,671.4	-	-
24	10	16,777,216	4,540,386	27%	331.6	101,348.6	-	-
24	12	16,777,216	9,740,686	58%	672.6	-	-	-

Occupation mémoire (Ko) - RS1-1												
Modèle					E-Sh		PR-Sh		FR-Sh		HBF	
N_p	N_r	$\|\hat{S}\|$	$\|S\|$	%	des	exec	des	exec	des	exec	mat	exec
20	1	1,048,576	21	0.002%	8,207	35,164	15	39,776	15	2,380	0	2,084
20	4	1,048,576	6,196	0.6%	8,207	35,164	63	40,040	63	2,816	640	2,832
20	6	1,048,576	60,460	5.8%	8,207	35,164	487	42,400	487	6,428	8,755	11,792
20	8	1,048,576	263,950	25.2%	8,207	35,164	2,077	51,128	2,077	19,932	48,272	54,484
20	10	1,048,576	616,666	58.8%	8,207	35,164	4,833	66,292	4,833	43,360	-	-
20	20	1,048,576	1,048,576	100%	8,207	35,164	8,207	84,860	8,207	72,052	-	-

Modèle					FR-Sh		HBF	
N_p	N_r	$\|\hat{S}\|$	$\|S\|$	%	des	exec	mat	exec
24	1	16,777,216	25	0.0001%	21	6,252	0	2,104
24	4	16,777,216	12,952	0.08%	122	7,116	1,354	3,668
24	6	16,777,216	190,051	1.1%	1,506	18,896	28,030	33,112
24	8	16,777,216	1,271,626	7.6%	9,956	90,712	-	-
24	10	16,777,216	4,540,386	27%	35,493	307,768	-	-
24	12	16,777,216	9,740,686	58%	76,120	424,808	-	-

Les résultats observés avec $N_p = 20$ confirment nos observations précédentes ($N_p = 16$). Lorsqu'on peut l'utiliser, c'est l'algorithme HBF qui est le meilleur au niveau de la durée d'exécution. C'est cependant le premier à être limité : lorsque le modèle possède une forte proportion d'états accessibles, on ne peut plus l'utiliser ($N_r \geq 10$).

On constate également que la durée de génération de la matrice HBF est relativement élevée, ce qui rend l'algorithme PR-Sh le plus performant (somme des durées) pour $4 \leq N_r \leq 8$. L'algorithme PR-Sh utilise cependant beaucoup de structures mémoires intermédiaires, ce qui limite son application sur les gros modèles. Son utilité réside dans l'accélération qu'il apporte pour la résolution de modèles avec peu d'états accessibles qui ne sont pas trop gros (il est alors meilleur que FR-Sh). Ainsi, pour $N_p = 20$, PR-Sh est meilleur que FR-Sh au niveau de la durée d'exécution.

Quant à l'algorithme E-Sh, c'est le plus approprié pour les modèles avec une forte proportion d'états accessibles. C'est l'algorithme le plus performant lorsque $N_r \geq 10$.

Lorsque $N_p = 24$, seuls les algorithmes FR-Sh et HBF peuvent encore aboutir. On atteint cependant les limites de l'algorithme HBF rapidement. Déjà, lorsque $N_r \geq 4$, le temps de génération de la matrice en format HBF est très long et rend l'algorithme HBF moins performant en durée d'exécution que l'algorithme FR-Sh. De plus, la place mémoire occupée par la matrice HBF est largement supérieure à celle occupée par le descripteur, et pour $N_r \geq 6$, FR-Sh est également plus performant au niveau de la mémoire utilisée à

l'exécution. Lorsque $N_r \geq 8$, on ne peut même plus générer la matrice. L'algorithme HBF ne peut plus être utilisé.

Les limites de l'algorithme FR-Sh sont également atteintes pour les modèles comportant une forte proportion d'états accessibles. En effet, lorsque $N_r \geq 12$, les besoins en mémoire de l'algorithme FR-Sh deviennent également trop importants, et la résolution ne peut plus avoir lieu. On retrouve ici le fait que l'algorithme FR-Sh exploite le fait qu'il puisse y avoir peu d'états accessibles dans le modèle. Si le modèle comporte beaucoup d'états accessibles, FR-Sh devient inefficace et inutilisable.

II.3.2. MÉTHODE AVEC PERMUTATION

Nous avons effectué une nouvelle série de tests sur le même modèle, mais en utilisant les algorithmes avec permutation. Nous nous sommes limités au cas où $N_r = 16$ pour observer le comportement des algorithmes, sachant que cette méthode est moins performante que celle sans permutation pour ce modèle, car toutes les évaluations doivent avoir lieu. On rajoute donc des permutations sans supprimer d'évaluations de fonctions.

Nous ne présentons pas ici les résultats correspondant à l'algorithme HBF, qui n'ont pas changé. Les données relatives au temps de compilation et à la taille du descripteur restent également identiques. Nous présentons les résultats concernant la durée d'exécution et l'occupation mémoire dans le même tableau.

Durée d'exécution (sec) et Occupation mémoire (Ko) - RS1-1										
Modèle					E-Sh		PR-Sh		FR-Sh	
N_p	N_r	$\lvert\hat{S}\rvert$	$\lvert S\rvert$	%	sec	Ko	sec	Ko	sec	Ko
16	1	65,536	17	0.03%	32.2	4,208	29.5	4,496	29.3	2,144
16	4	65,536	2,517	3.8%	132.4	4,208	124	4,636	123.1	2,292
16	6	65,536	14,893	22.7%	209.8	4,208	220	5,312	217.7	2,972
16	8	65,536	39,203	59.8%	277.9	4,208	367	6,656	368.1	4,304
16	10	65,536	58,651	89.5%	345.8	4,212	545.7	7,724	542.7	5,372
16	12	65,536	64,839	98.9%	392	4,212	659.8	8,060	656.6	5,708
16	16	65,536	65,536	100%	26.8	4,212	99	7,344	379.5	6,536

Sur ces résultats, on constate que les algorithmes PR-Sh et FR-Sh permettent un gain de temps à l'exécution beaucoup plus faible que dans la méthode sans permutation. Ceci est lié au fait que la permutation de vecteurs réduits engendre un surcoût par rapport à la permutation de vecteurs étendus. On remarque également que l'algorithme FR-Sh est plus performant que PR-Sh dans ces exemples, à la fois au niveau du temps d'exécution et de la place mémoire utilisée. Lorsqu'on utilise les méthodes avec permutation sans événements synchronisants, on emploie dorénavant systématiquement l'algorithme FR-Sh à la place de PR-Sh.

II.4. RS2

Il est possible de représenter le modèle de partage de ressources à l'aide d'un réseau d'automates stochastiques sans utiliser de fonctions. C'est ce que nous avons montré avec le modèle **RS2** présenté dans la section IV.1.2 du chapitre 2 (page 55). Cette représentation complexifie le modèle et rend la résolution du modèle plus longue, mais elle est intéressante pour tester les algorithmes sans évaluation de fonction, et qui utilisent des vecteurs intermédiaires (traitement des événements synchronisant).

Il n'y a pas de fonctions dans cet exemple. La méthode sans permutation est donc identique à la méthode avec permutation.

Durée d'exécution (sec) - RS2												
Modèle					E-Sh		PR-Sh		FR-Sh		HBF	
N_p	N_r	$\|\hat{S}\|$	$\|S\|$	%	comp	exec	comp	exec	comp	exec	géné	exec
16	1	131,072	17	0.01%	33.6	13.6	0.07	6.1	0.07	2.2	3.8	0
16	4	327,680	2,517	0.77%	84.3	85.4	0.7	40.1	0.7	11.1	1,231.1	0.04
16	6	458,752	14,893	3.25%	116.1	174.9	3.9	101.3	3.9	47.9	10,290.5	0.7
16	8	589,824	39,203	6.65%	149.1	296.5	9.9	219.8	9.9	150.3	37,502.4	3.2
16	10	720,896	58,651	8.14%	182.6	563.1	15.3	449.8	15.3	350.9	70,182.6	7.8
16	12	851,968	64,839	7.61%	215.6	1,002.5	16.8	774.6	16.8	603.5	89,655.6	13.4
16	16	1,114,112	65,536	5.88%	283.3	1,861.5	17	1,350	17	918.2	116,837.4	19.7

Occupation mémoire (Ko) - mutex2												
Modèle					F		SF		S		HBF	
N_p	N_r	$\|\hat{S}\|$	$\|S\|$	%	des	exec	des	exec	des	exec	mat	exec
16	1	131,072	17	0.01%	1,210	6,480	187	7,928	187	2,368	0	2,104
16	4	327,680	2,517	0.77%	2,747	12,672	207	21,784	207	2,628	255	2,404
16	6	458,752	14,893	3.25%	3,772	16,800	304	37,440	304	3,628	2,086	4,432
16	8	589,824	39,203	6.65%	4,796	20,928	494	59,860	494	5,564	6,756	9,484
16	10	720,896	58,651	8.14%	5,821	25,060	647	84,604	647	7,116	11,350	14,380
16	12	851,968	64,839	7.61%	6,845	29,188	696	112,564	696	7,628	13,085	16,212
16	16	1,114,112	65,536	5.88%	8,894	37,452	702	179,872	702	7,780	13,312	16,460

On constate tout d'abord sur ces tests une réelle inefficacité de l'algorithme PR-Sh qui est moins bon à la fois au niveau place mémoire et au niveau du temps d'exécution que l'algorithme FR-Sh.

La différence de place mémoire utilisée est expliquée par le fait que l'algorithme PR-Sh utilise des structures intermédiaires en format étendu, alors que FR-Sh n'utilise que des structures réduites.

Au niveau du temps d'exécution, on a vu que l'algorithme PR-Sh n'est pas efficace lorsqu'il y a des vecteurs intermédiaires, car on ne sait pas quels éléments du résultat n'ont pas besoin d'être calculés, et on effectue donc $|\hat{S}|$ multiplications de tranches de vecteurs par une colonne de matrice. En revanche, l'algorithme FR-Sh effectue des tests pour savoir si un z_{in} contient au moins un élément non nul avant d'évaluer la matrice, puis de multiplier le z_{in} par la matrice. Ces tests permettent une diminution du nombre d'évaluations et de multiplications, ce qui entraîne un gain de temps par rapport à l'algorithme PR-Sh qui ne fait pas de tests.

Par la suite, nous n'utilisons plus l'algorithme PR-Sh lorsque le résultat est un vecteur intermédiaire. Nous utilisons à la place l'algorithme FR-Sh.

Pour s'assurer que la mémoire requise par le nouvel algorithme dépend vraiment de $|S|$, et non plus de $|\hat{S}|$ comme pour E-Sh, nous avons tracé des courbes qui donnent l'utilisation mémoire des algorithmes E-Sh et FR-Sh en fonction de la taille de l'espace d'états accessibles S (figure 5.2).

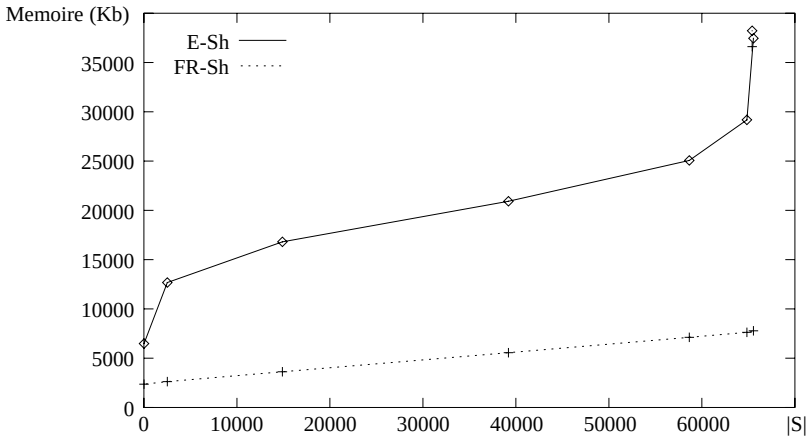


FIG. 5.2 – Occupation mémoire pour le modèle RS2

La mémoire utilisée par FR-Sh est bien proportionnelle à S, alors que ce n'est pas le cas pour E-Sh. Pour E-Sh, on obtient une droite si l'on trace la courbe d'occupation mémoire en fonction de $|\hat{S}|$, ce qui confirme que la mémoire utilisée est dans ce cas proportionnelle à $|\hat{S}|$. L'origine de la droite correspond à la mémoire minimum requise pour l'exécution de l'algorithme FR-Sh. Pour ce modèle, il s'agit de 2400 Kb.

L'utilisation d'événements synchronisant engendre souvent des modèles avec une faible proportion d'états accessibles. L'espace d'états produit doit en effet être agrandi par rapport au modèle **RS1**, mais l'espace des états accessibles reste le même. On ne teste donc ici que des modèles où l'algorithme FR-Sh est meilleur que l'algorithme E-Sh, car il y a toujours une faible proportion d'états accessibles.

Enfin, nous constatons sur ces exemples une réelle inefficacité de l'algorithme de génération de la matrice HBF lorsqu'il y a des événements synchronisant. Ainsi, pour $N_r \geq 4$, la génération de la matrice HBF est très longue en comparaison de la somme des durées de compilation et d'exécution pour les trois autres algorithmes. Ce phénomène est cependant lié à l'implémentation de l'algorithme de génération dans PEPS. Une génération plus efficace peut être obtenue à l'aide du logiciel MARCA [90]. L'inconvénient majeur de la méthode HBF vient du fait que la matrice générée occupe plus de place mémoire que le descripteur, et la mémoire à l'exécution de l'algorithme HBF est également supérieure à la mémoire à l'exécution de l'algorithme FR-Sh.

L'algorithme FR-Sh est donc le plus approprié lorsqu'il y a des événements synchronisant dans le modèle, à la fois au niveau de la mémoire et au niveau du temps d'exécution.

II.5. QN1

Le dernier modèle présenté est celui d'un réseau de files d'attente ouvert avec blocage et priorité **QN1**, présenté dans la section IV.2.1 du chapitre 2 (page 57). Il est composé de N files de capacité finie, et de $N-1$ classes de clients différents.

Les tests ont été effectués à l'aide de la méthode sans permutation. Nous faisons varier N et la capacité de la file N pour que la proportion d'états accessibles et la taille du modèle changent. Dans tous les exemples, pour $i = 1..N-1$, $C_i = 1$, $\lambda_i = 6$, $\mu_i = 9$ et $\mu_N^i = 9$.

De plus, pour ces tests, nous n'utilisons plus l'algorithme PR-Sh lorsque le résultat est un vecteur intermédiaire, car son inefficacité dans ce cas a déjà été montrée (résultats sur le modèle RS2). Nous utilisons donc à la place l'algorithme FR-Sh. La colonne R-Sh (*reduced Shuffle*) est donc obtenue en employant suivant le cas l'algorithme PR-Sh ou l'algorithme FR-Sh, pour obtenir les meilleures performances possibles.

Durée d'exécution (sec) - QN1												
Modèle					E-Sh		R-Sh		FR-Sh		HBF	
N	C_N	$\lvert\hat{S}\rvert$	$\lvert S\rvert$	%	comp	exec	comp	exec	comp	exec	géné	exec
8	2	279,936	4,608	1.6%	18.9	5,733.6	0.4	957.9	0.4	1,496.3	433.8	1.7
8	3	2,097,152	15,360	0.7%	147.2	68,048.9	1.4	10,583.5	1.4	15,343.2	11,109.8	11

Occupation mémoire (Ko) - QN1												
Modèle					E-Sh		R-Sh		FR-Sh		HBF	
N	C_N	$\lvert\hat{S}\rvert$	$\lvert S\rvert$	%	des	exec	des	exec	des	exec	mat	exec
8	2	279,936	4,608	1.6%	2,222	10,984	71	13,608	71	2,600	355	2,548
8	3	2,097,152	15,360	0.7%	16,419	68,236	155	111,240	155	3,912	1,237	3,592

Pour ce type de modèles, c'est l'algorithme FR-Sh qui est le plus performant au niveau de la mémoire parmi les trois algorithmes basés sur une représentation tensorielle du descripteur. En effet, les modèles obtenus comportent une faible proportion d'états accessibles du fait des événements synchronisant. L'utilisation d'un algorithme combinant PR-Sh et FR-Sh (R-Sh) permet d'obtenir une durée d'exécution encore meilleure que celle obtenue par l'algorithme FR-Sh. Les synchronisations ne réduisent plus les performances de ce nouvel algorithme R-Sh. En revanche, la place mémoire nécessaire pour appliquer cet algorithme est similaire à celle de PR-Sh ; c'est donc une limitation à l'application de ce nouvel algorithme.

Lorsque $N = 12$ et $C_N = 2$, on arrive à $|\hat{S}| = 362,797,056$ et $|S| = 159,744$. Ce modèle avec un très grand nombre d'états peut être résolu uniquement à l'aide de l'algorithme FR-Sh.

Les résultats confirment également la grande rapidité d'exécution de l'algorithme HBF une fois la matrice générée. Ainsi, pour le modèle avec $N = 8$ et $C_N = 2$, il suffit de 1.7 secondes à cet algorithme pour aboutir. La taille de la matrice HBF est cependant supérieure

à la taille du descripteur (365 Ko contre 71 Ko pour le descripteur). De plus, le temps de génération de la matrice reste un obstacle à l'aboutissement de cet algorithme avec PEPS.

II.6. CONCLUSION

Nous avons présenté dans le chapitre 4 l'algorithme du shuffle classique E-Sh, ainsi que deux variantes exploitant le fait qu'un grand nombre d'états peuvent ne pas être accessibles. Les tests comparatifs effectués nous permettent de dresser le bilan suivant :

- **L'algorithme E-Sh** (*extended shuffle*) est le plus efficace lorsqu'il y a une grande proportion d'états accessibles. Ainsi, lorsque plus de la moitié des états sont accessibles, on utilisera de préférence cet algorithme.
- **L'algorithme PR-Sh** (*partially reduced shuffle*) exploite le fait qu'un grand nombre de modèles possède une faible proportion d'états accessibles pour améliorer les performances de la multiplication vecteur-descripteur lorsqu'il n'y a pas d'événements synchronisant. Cet algorithme utilise cependant des structures intermédiaires en format étendu (structures de taille $\hat{S}$), ce qui limite son application sur des gros modèles. De plus, lorsqu'il y a des événements synchronisant (algorithme avec obtention d'un vecteur intermédiaire), les performances sont mauvaises par rapport à l'algorithme FR-Sh, nous ne l'utilisons donc pas dans ce cas là. Le gain de temps par rapport à l'algorithme classique E-Sh est alors très élevé pour les modèles à faible proportion d'états accessibles, et avec un nombre d'états raisonnable (de l'ordre du million). La principale utilité de PR-Sh réside donc dans les modèles pour lesquels la génération de la matrice HBF est coûteuse, mais où le modèle est suffisamment petit pour que l'algorithme PR-Sh puisse tout de même être employé.
- **L'algorithme FR-Sh** (*fully reduced shuffle*) n'utilise que des structures en format réduit. On arrive donc à traiter de très gros modèles avec cet algorithm, modèles qu'aucun autre algorithme ne peut traiter. On observe cependant une perte de temps à l'exécution par rapport à l'algorithme PR-Sh lorsqu'il n'y a pas de vecteurs intermédiaires. C'est le prix à payer pour pouvoir traiter de gros modèles.
- **L'algorithme HBF**, qui génère le descripteur sous forme d'une matrice creuse, est très performant. Sa principale limitation vient de la génération de la matrice HBF, qui peut être très longue, voire ne pas aboutir faute de place mémoire disponible. La place mémoire occupée par cette matrice est de plus supérieure à la place occupée par le descripteur en format tensoriel, et la mémoire à l'exécution est souvent supérieure à celle occupée par l'algorithme FR-Sh. Lorsqu'on peut l'utiliser (notamment sur des modèles avec très peu d'états accessibles), c'est l'algorithme HBF qui est le plus efficace.

Tous les algorithmes testés ont des avantages et des inconvénients, et conviennent particulièrement bien pour résoudre une certaine catégorie de modèles. Pour exploiter les particularités de chaque algorithme, nous prévoyons de développer une heuristique qui choisit automatiquement l'algorithme le plus approprié. Ainsi, l'utilisateur peut soit choisir l'algorithme qu'il souhaite appliquer, soit laisser au programme le soin de décider à sa place

l'algorithme le plus approprié au modèle à résoudre, d'après les paramètres du modèle (proportion d'états accessibles, taille du modèle, ...).

Dans le formalisme SAN, l'utilisation des fonctions permet de réduire la taille de l'espace produit. L'utilisation d'algèbre tensorielle généralisée ([41, 6]) autorise des opérations tensorielles sur des matrices fonctionnelles. Cependant, le coût des évaluations de matrices est élevé, nous cherchons donc à en réduire le nombre. Les techniques de réordonnancement d'automates développées dans le chapitre 4 permettent de réduire le nombre d'évaluations de matrices dans l'algorithme du shuffle. Le groupement algébrique d'automates ([41]) est une autre technique qui peut être utilisée pour réduire le nombre d'évaluations de fonctions. Cette technique n'a pas été présentée car l'utilisation de vecteurs réduits ne change en rien la manière de procéder.

Enfin, les nouveaux algorithmes ont été comparés uniquement à l'algorithme du shuffle classique E-Sh, et à l'algorithme qui calcule le générateur en format HBF. Une comparaison avec d'autres algorithmes classiques, et notamment les algorithmes basés sur les réseaux de Pétri [20, 10, 66] n'a pas été effectuée ici. On envisage également de comparer nos algorithmes avec ceux basés sur les diagrammes de matrices. En fait, Ciardo propose une comparaison entre ces deux approches dans [31], mais il n'utilise pas les algorithmes les plus récents qui ont été développés pour l'approche Kronecker. Ses résultats prouvent que les diagrammes de matrices sont plus performants que l'algorithme Act-Sh-JCB de [20], mais il est maintenant nécessaire d'effectuer de nouvelles expérimentations avec FR-Sh pour avoir une comparaison avec un algorithme qui améliore à la fois la place mémoire nécessaire et les temps d'exécution lorsqu'il y a une faible proportion d'états accessibles.

Ces améliorations à l'algorithme du shuffle nous permettent ainsi d'analyser des systèmes parallèles et distribués avec un espace d'états de plus en plus grand, et ceci nous semble d'une importance primordiale dans un contexte où de tels systèmes ne font que se complexifier.

III. AGRÉGATION DE SANS AVEC RÉPLICATION

Nous avons effectué dans la section précédente une étude comparative des différents algorithmes de multiplication vecteur-descripteur présentés dans le chapitre 4. Le développement de nouveaux algorithmes a ainsi permis de résoudre des modèles que l'on ne pouvait pas traiter précédemment.

Cependant, la plupart de ces modèles présentent des symétries, et de bien meilleurs résultats sont obtenus en commençant par agréger le réseau d'automates stochastiques, puis en travaillant sur le modèle agrégé. L'espace d'états agrégés est alors d'une taille souvent négligeable par rapport à la taille de l'espace d'états initial, ce qui permet d'obtenir de très bons résultats en terme de mémoire et de temps d'exécution.

Les résultats sont calculés dans les mêmes conditions que précédemment (section II.1) dans un souci de comparaison, à l'aide du logiciel PEPS (section I). Le temps indique en revanche le temps utilisateur mis pour exécuter une seule itération (dans la section précédente, c'est le temps total d'exécution qui est indiqué). L'algorithme utilisé est toujours E-Sh. Un temps de $0\ sec.$ signifie que le temps d'exécution est négligeable (inférieur à $10^{-4}\ sec.$). Un temps de $-$ signifie que PEPS n'a pas pu résoudre le modèle à cause du trop grand nombre d'états.

Modèle	SAN initial			chaîne de Markov agrégée		
	$\lvert\hat{S}\rvert$	$\lvert S\rvert$	Temps (sec.)	$\lvert\hat{S}_{agg}\rvert$	$\lvert S_{agg}\rvert$	Temps (sec.)
RS1-1 $N_p = 20$, $N_r = 10$	$1,048,576$	$616,666$	15.7	21	11	0.000
RS1-1 $N_p = 10,000$, $N_r = 8,000$	$2^{10,000}$	-	-	$10,001$	$8,001$	0.004
RS1-2 $N_r = 10$ $R_1 = 5$, $R_2 = 15$	$1,048,576$	$616,666$	15.7	96	51	0.000
RS3 $N_p = 20$, $N_r = 10$	$\approx 7 * 10^9$	$616,667$	-	462	12	0.000

Ces résultats confirment le fait que l'agrégation produit toujours une réduction de l'espace d'états significative, comme cela avait été montré de façon théorique dans la section III du chapitre 3.

En général, après agrégation, le temps requis pour calculer des indices de performances pour le modèle devient négligeable, si l'on considère les modèles que l'on pouvait traiter auparavant (et notamment les modèles traités dans la section précédente). Ainsi, alors qu'il était difficile de résoudre un problème de partage de ressources avec 24 clients, on n'a maintenant aucune difficulté à traiter un modèle avec plus de $10,000$ clients !

L'identification des répliquas dans un modèle et l'agrégation de SANs permettent donc de résoudre des problèmes qui étaient auparavant intraitables à cause du grand nombre d'états. C'est le cas du modèle RS1-1 avec $N_p = 10,000$, et du partage de ressources avec pannes (modèle RS3).

CHAPITRE

Conclusions et perspectives

6

Nous dressons dans une première section un bilan des travaux effectués au cours de cette thèse, en rappelant les principaux résultats et les moyens mis en oeuvre pour les obtenir.

Ensuite, nous évoquons les nombreuses perspectives ouvertes par ces travaux, et nos pistes de recherche.

I. CONCLUSIONS

L'apport scientifique de cette thèse se situe sur plusieurs plans.

Pour la modélisation des systèmes (phase d'**abstraction**), nous proposons le formalisme des réseaux d'automates stochastiques, adapté aux systèmes à grand espace d'états. La définition des réseaux d'automates stochastiques avec réplication permet d'aller encore plus loin dans la taille des espaces d'états. Un formalisme de réseaux d'automates stochastiques pour les modèles à temps discret est également proposé.

Pour calculer des indices de performances (phase de **résolution**), nous utilisons des méthodes itératives et proposons une amélioration de la multiplication vecteur-descripteur, toujours dans le but de pouvoir traiter des modèles à grand espace d'états.

I.1. ABSTRACTION

La phase d'abstraction du processus de modélisation consiste à créer un modèle à partir d'un système réel (cf figure 1.1 page 13). Dans ces travaux, nous nous sommes focalisés sur le formalisme des *réseaux d'automates stochastiques* (SANs). Ce formalisme permet la modélisation structurée de systèmes à grand espace d'états, en représentant le système sous la forme d'un ensemble de composants indépendants qui interagissent. Chaque composant est un automate traditionnel, auquel on rajoute des comportements stochastiques et des mécanismes de synchronisation.

Le formalisme des SANs à **temps continu** est largement développé depuis une vingtaine d'années. Nous avons présenté ce formalisme d'une nouvelle manière en identifiant précisément les événements qui apportent un changement d'état au système. C'est le recul pris par rapport à ce formalisme et les besoins de simplifier la modélisation de systèmes qui nous ont amené à donner de nouvelles définitions plus cohérentes des SANs.

Nous avons alors pu introduire une extension logique pour traiter des modèles à **temps discret**, en précisant bien la sémantique à utiliser en cas de conflit. Des travaux avaient déjà été effectués à ce sujet, mais nous nous sommes rendus compte que certains points du formalisme de SAN à temps discret n'étaient pas assez précis, et le comportement du système n'était pas totalement décrit. Nous avons de plus proposé un algorithme pour générer la chaîne de Markov à temps discret sous-jacente, même si pour l'instant nous ne disposons pas de représentation tensorielle du générateur de cette chaîne de Markov.

Notre intérêt étant porté sur les systèmes à grand espace d'états, nous avons tout naturellement cherché à réduire cet espace pour pouvoir traiter des systèmes de plus en plus complexes. Pour ce faire, nous avons utilisé des techniques d'**agrégation exacte**, déjà développées dans la littérature mais rarement employées sur des formalismes de haut niveau comme les réseaux d'automates stochastiques.

Nous avons ainsi exploité le fait que les systèmes sont généralement constitués d'un grand nombre de composants identiques, et défini une classe de SANs propices à une agrégation, les **réseaux d'automates stochastiques avec réplication**. La part la plus ardue du travail a consisté à démontrer que le théorème de Rosenblatt d'agrégation exacte pouvait être appliqué sur les SANs avec réplication, puis à montrer que la matrice de la chaîne de Markov agrégée pouvait s'exprimer sous forme tensorielle.

I.2. RÉSOLUTION

Nous ne nous sommes pas contentés de travailler sur la phase d'abstraction, il a également été nécessaire de travailler sur les techniques de résolution pour pouvoir traiter des systèmes complexes.

Certaines méthodes numériques, et notamment les méthodes itératives, sont particulièrement bien adaptées à l'étude de systèmes à grand espace d'états. La principale difficulté pour l'application de ces méthodes réside dans la multiplication d'un vecteur de probabilité par le descripteur du SAN. Cette simple multiplication vecteur-matrice est rendue délicate par la taille des données à traiter. Les vecteurs comportent en effet une entrée par état du système, et lorsque le nombre d'états est élevé, il devient difficile de stocker ces vecteurs en mémoire.

La structure tensorielle du descripteur du SAN permet l'utilisation d'algorithmes adaptés, tels le classique algorithme du shuffle. Cependant, cet algorithme utilise des vecteurs *étendus* (de la taille de l'espace d'états produit $\hat{S}$). Nous avons donc développé de nouvelles versions de cet algorithme, en exploitant le fait que dans de nombreux modèles, peu d'états sont accessibles. Nous utilisons ainsi des vecteurs *réduits* (de la taille de l'espace d'états accessibles S), ce qui permet un gain en mémoire et en temps d'exécution significatifs pour une classe de modèles.

Ces algorithmes ont été implémentés dans le logiciel PEPS 2003, et de nombreux résultats numériques ont été obtenus par ce moyen. Nous avons également été amené durant cette thèse à effectuer d'autres modifications sur le logiciel, notamment en identifiant des

problèmes de fonctionnement dans PEPS et des algorithmes inefficaces (par exemple, l'algorithme de génération de la matrice en format HBF).

Grâce aux résultats numériques obtenus, nous avons pu comparer les algorithmes développés aux algorithmes classiques, et montrer l'intérêt des nouvelles méthodes et algorithmes développés pour l'évaluation des performances des systèmes informatiques à grand espace d'états.

II. PERSPECTIVES

De nombreuses perspectives ont été ouvertes suite à ces travaux.

Nous présentons tout d'abord les perspectives de développement autour du formalisme des réseaux d'automates stochastiques, que ce soit au niveau de la modélisation ou de la résolution.

Un des grands axes de recherche envisagé consiste à comparer les méthodes et algorithmes développés pour le formalisme des réseaux d'automates stochastiques avec les techniques couramment utilisées pour d'autres formalismes, et notamment les réseaux de Petri. Nous détaillerons cet aspect dans la section II.2.

Enfin, nous nous intéressons également aux travaux effectués dans des domaines voisins de l'évaluation de performances, et notamment nous envisageons de travailler sur le lien entre la vérification de systèmes et l'évaluation de performances (section II.3).

II.1. RÉSEAUX D'AUTOMATES STOCHASTIQUES

De nombreuses recherches restent à effectuer sur le formalisme des réseaux d'automates stochastiques.

II.1.1. RÉSEAUX D'AUTOMATES STOCHASTIQUES À TEMPS DISCRET

Nous avons dans cette thèse exposé un formalisme de réseaux d'automates stochastiques à temps discret, et présenté un algorithme de génération de la chaîne de Markov sous-jacente, mais l'étude de ce nouveau formalisme reste un domaine de recherche pleinement ouvert.

Déjà, nous n'avons pas proposé pour l'instant d'**expression tensorielle** pour représenter le descripteur d'un SAN à temps discret. Ceci nécessite la définition de nouveaux opérateurs tensoriels tenant compte des priorités. Chaque élément du générateur ainsi obtenu doit alors correspondre à une probabilité calculée par l'algorithme de génération de la chaîne de Markov.

Ensuite, un gros travail reste à faire sur les **méthodes de résolution** sur ce formalisme. La représentation du descripteur sous forme d'expression tensorielle devrait per-

mettre d'adapter les méthodes classiques, et notamment d'utiliser l'algorithme du shuffle. Cependant, la présence des priorités et la modification probable des opérateurs tensoriels risque de nécessiter de grands changements par rapport aux méthodes utilisées sur les réseaux d'automates stochastiques à temps continu. Ceci constitue donc un domaine de recherche totalement ouvert, mais les travaux sur le sujet ne pourront commencer que lorsque la phase d'abstraction, et notamment la définition d'un descripteur en format tensoriel, aura été terminée.

II.1.2. Réseaux d'automates stochastiques avec réplication

Au niveau des réseaux d'automates stochastiques avec réplication, nous avons proposé une définition formelle et un algorithme d'agrégation. L'algorithme est basé sur le théorème de Rosenblatt, et montre que l'on peut agréger l'espace d'états pour former une chaîne de Markov agrégée.

En démontrant que la matrice de cette chaîne de Markov agrégée peut s'exprimer sous forme tensorielle, nous avons suggéré l'idée de définir un réseau d'automates stochastiques équivalent par agrégation séparée de chaque groupe d'automates identiques. Ce SAN n'a cependant pas été défini ni même introduit. Pour prouver son existence et le fait que son comportement est identique au comportement du SAN initial, il reste à en donner une définition formelle.

A partir de là, on peut envisager de simplifier les démonstrations effectuées en allégeant les notations, et en partant directement du SAN équivalent. Il s'agit de prouver que l'on peut agréger séparément chaque groupe d'automates pour former un automate stochastique équivalent. Cet automate équivalent peut comporter des événements qui le synchronisent avec l'extérieur (d'autres groupes d'automates), ainsi que des taux fonctionnels. Tous ces automates mis en réseau forment alors le SAN équivalent.

Une extension peut être envisagée pour agréger des réseaux d'automates stochastiques à temps discret avec réplication. Nous nous sommes limités dans ces travaux à une approche en temps continu.

II.1.3. Interface graphique PEPS

Enfin, pour rendre l'utilisation des SANs plus conviviales pour les utilisateurs, nous prévoyons l'intégration d'une interface graphique dans PEPS.

Cette interface, écrite en Java, est en cours de développement par Mathieu Le Coz, au laboratoire PRiSM (Université de Versailles). La figure 6.1 présente une vue générale de l'interface, et la figure 6.2 montre comment l'on peut facilement éditer et dessiner chaque automate du SAN. On peut alors générer le fichier texte qui décrit le SAN en format PEPS.

Des modifications restent à apporter à cette interface pour mieux l'adapter à la nouvelle interface de PEPS, notamment en définissant systématiquement chaque événement.

De plus, nous envisageons de donner la possibilité à l'utilisateur d'appeler directement les fonctionnalités de PEPS en rajoutant des menus à l'interface graphique. Ceci permettra

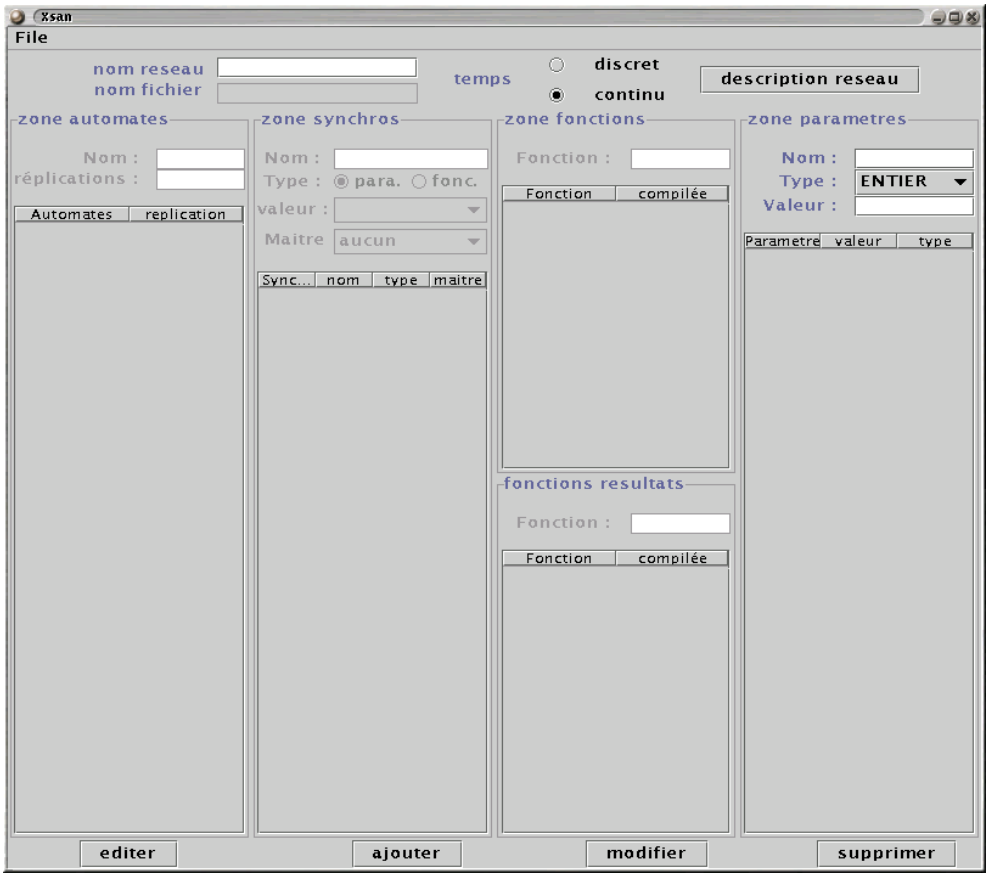


FIG. 6.1 – Interface graphique PEPS, vue générale

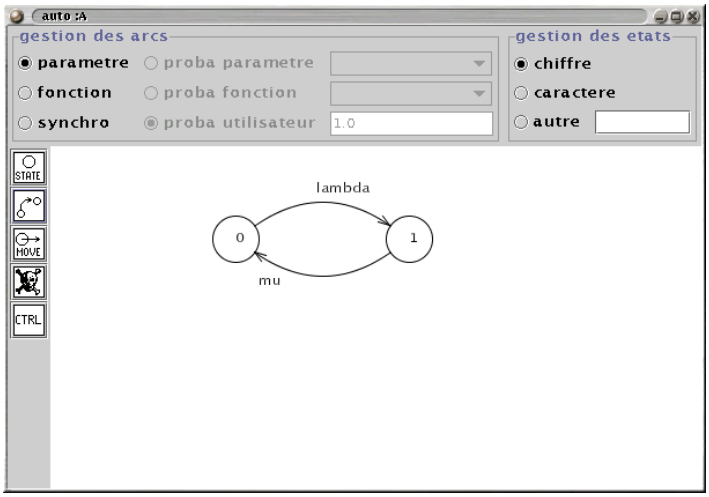


FIG. 6.2 – Interface graphique PEPS, dessin d'un automate

de rendre également plus conviviale l'utilisation de PEPS, qui fonctionne actuellement par l'intermédiaire d'une interface textuelle.

II.2. COMPARAISON AVEC D'AUTRES FORMALISMES

Nous avons dans cette thèse proposé des méthodes et algorithmes basées sur le formalisme des réseaux d'automates stochastiques.

De nombreux travaux similaires existent. La plupart sont basés sur des formalismes voisins des SANs, pour lesquels des techniques sont développées pour évaluer les performances de systèmes à grand espace d'états.

Une des perspectives ouvertes par ces travaux consiste donc à comparer d'une part la puissance de modélisation des SANs par rapport à d'autres formalismes tels les réseaux de Petri, et d'autre part à comparer les méthodes et algorithmes sur des modèles identiques.

Pour comparer les différents formalismes, nous avons débuté un programme d'échange d'exemples avec G. Ciardo et S. Donatelli (projet NSF), et nous traduisons ainsi des modèles de réseau de Petri en modèles SANs. L'idée est d'arriver à systématiser le passage d'un modèle SAN à un modèle réseau de Petri de manière à pouvoir utiliser l'un ou l'autre formalisme, suivant le modèle et les techniques de résolution dont on dispose.

Pour comparer les méthodes de résolution, il est nécessaire d'effectuer d'une part une analyse théorique fine des algorithmes, et d'autre part d'obtenir des résultats numériques à partir de logiciels qui implémentent différentes techniques, mais en utilisant la même architecture.

Ce travail de comparaison nécessite du temps et une grande maîtrise de toutes les techniques les plus récentes. Aucune étude comparative sérieuse n'a pu être menée à notre connaissance, par manque d'échange entre les différentes communautés. L'intérêt d'une telle étude est énorme, elle permettra en effet de mettre en évidence les problèmes liés à chaque approche, et de tirer profit de l'analyse de ces problèmes pour améliorer chaque méthode.

II.3. DOMAINES CONNEXES

D'autres domaines de recherche sont très liés à l'évaluation de performances, et notamment la vérification de systèmes.

Nous envisageons de développer un outil logiciel qui traduise un modèle UML en un modèle SAN, dans le but de combiner à la fois la vérification du système et l'évaluation de ces performances dans un unique logiciel. L'obtention d'un tel logiciel est un véritable défi que l'on peut se lancer, et des travaux en cours sur la traduction d'un formalisme à un autre nous montrent que cette piste de recherche peut apporter beaucoup pour l'évaluation de performances (nouvelles idées, nouvelles techniques, nouveaux algorithmes).

Glossaire

accessible : un état est accessible si c'est un successeur d'un état accessible, sachant qu'un état initial donné est accessible (définition 4 page 39).

automate équivalent : graphe états-transitions, constitué des états globaux du système, et dont les transitions sont des taux de franchissement (page 32).

automate stochastique : graphe état-transitions, constitué d'un ensemble d'états locaux et d'un ensemble de transitions (page 37).

espace d'états produits d'un réseau d'automates stochastique : c'est l'ensemble des états globaux du réseau d'automates stochastique (page 37).

état global d'un réseau d'automates stochastique : vecteur d'états locaux, un état local par automate du réseau d'automates stochastiques (page 37).

état intermédiaire : état obtenu après qu'un événement ait eu lieu dans un modèle à temps discret (définition 14 page 46).

état local d'un automate stochastique : l'un des états locaux (page 37).

état successeur par l'événement e : état d'arrivée possible lorsque l'événement e a lieu (temps continu, définition 3 page 39).

état vide : état fictif pour les modèles à temps discret, qui permet qu'un événement soit possible depuis un état sans être réalisable (définition 12 page 45).

événement : défini en temps continu par un taux de franchissement, un ensemble d'automates impliqués et un automate maître (définition 1 page 38), et en temps discret par une probabilité de transition, une priorité et un ensemble d'automates impliqués (définition 11 page 45).

événement local : événement qui ne concerne qu'un seul automate (définition 1 page 38, définition 11 page 45).

événement possible : un événement est possible dans un état donné s'il se peut qu'il ait lieu (définition 3 page 39, définition 14 page 46).

événement qui a lieu : un événement a lieu si les transitions correspondantes se déclenchent (définition 3 page 39, définition 14 page 46).

événement réalisable : un événement est réalisable s'il est possible avec un taux de franchissement non nul (temps continu, définition 3 page 39), ou s'il est possible avec une probabilité de transition non nulle, s'il mène à un état différent de l'état vide (temps discret, définition 14 page 46).

événement synchronisant : événement qui concerne au moins deux automates (définition 1 page 38, définition 11 page 45).

facteur normal : produit tensoriel d'une matrice carrée par une matrice identité (page 25).

liste d'états accessibles pondérés par des probabilités : liste de couples (état, probabilité) (définitions 15 et 16 page 47).

réseau d'automates stochastiques : chaîne de Markov à N composantes, comportant N automates stochastiques et un ensemble d'événements (page 37).

taux de franchissement : définition 1 page 38.

transition : définie par un état de départ, un événement et une probabilité de routage (définition 2 page 38, définition 13 page 45).

vecteur étendu : vecteur de la taille de l'espace produit $\hat{S}$ (chapitre 4).

vecteur intermédiaire : définition 20 page 104

vecteur de probabilité : définition 20 page 104

vecteur réduit : vecteur de la taille de l'espace d'états accessibles S (chapitre 4).

Table des figures

Bibliographie

[1] S. Alouf, F. Huet, and P. Nain. Forwarders vs. centralized server : an evaluation of two approaches for locating mobile agents. *Performance Evaluation*, 49 :299–319, Sept. 2002.

[2] K. Atif. *Modélisation du parallélisme et de la synchronisation*. PhD thesis, Institut National Polytechnique de Grenoble, France, 1992.

[3] F. Bause, P. Buchholz, and P. Kemper. A Toolbox for functionnal and quantitative analysis of DEDS (extended abstract). In *Proc. 10th. int. conf. Computer Performance evaluation, Modelling Techniques and Tools, LNCS 1469, Springer*, 1998.

[4] A. Benoit, L. Brenner, P. Fernandes, and B. Plateau. Aggregation of Stochastic Automata Networks with replicas. *Submitted to NSMC'03*, 2003.

[5] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, and W.J. Stewart. The PEPS Software Tool. *Submitted to Performance TOOLS 2003*, 2003.

[6] A. Benoit, P. Fernandes, B. Plateau, and W.J. Stewart. On the Benefits of Using Functional Transitions in Kronecker Modelling. *Submitted to Performance Evaluation*, 2002.

[7] A. Benoit, B. Plateau, and W.J. Stewart. Memory Efficient Iterative Methods for Stochastic Automata Networks. Technical report, Rapport de recherche INRIA n. 4259, France, Sept. 2001.

[8] A. Benoit, B. Plateau, and W.J. Stewart. Memory-efficient Kronecker algorithms with applications to the modelling of parallel systems. *To appear in PMEO-PDS'03*, 2003.

[9] M. Bozga and O. Maler. On the Representation of Probabilities over Structured Domains. In N. Halbwachs and D. Peled, editors, *Proc. CAV'99*, volume 1633 of *LNCS*, pages 261–273. Springer, June 1999.

[10] P. Buchhloz. Hierarchical structuring of Superposed GSPNs. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models PNPM'97*, pages 81–90, Saint Malo, France. IEEE CS Press, 1997.

[11] P. Buchholz. Aggregation and reduction techniques for hierarchical GCSPNs. In *Proc. 5th Int. Workshop on Petri Nets and Performance Models (PNPM'93), IEEE CS-Press*, pages 216–225, 1993.

[12] P. Buchholz. Hierarchical Markovian Models - Symmetries and Reduction. In *R.Pooley and J.Hillston, editors, Performance Evaluation'92 : Modelling Techniques and Tools, Edinburgh University Press*, pages 234–246, 1993.

[13] P. Buchholz. A Class of Hierarchical Queueing Networks and Their Analysis. *Queueing Systems*, 15 :59–80, 1994.

[14] P. Buchholz. Exact and ordinary lumpability in finite Markov chains. *Journal of Applied Probability*, 31 :59–75, 1994.

[15] P. Buchholz. Equivalence relations for Stochastic Automata Networks. In *Proc. 2nd int. workshop on the numerical solution of Markov chains, Kluwer ed. Boston, Masachussetts*, 1995.

[16] P. Buchholz. An aggregation-disaggregation algorithm for stochastic automata networks. *Probability in the Engineering and Informational Sciences*, 11 :229–253, 1997.

[17] P. Buchholz. An adaptative aggregation-disaggregation algorithm for hierarchical Markovian models. *European Journal of Operational Research*, 116 :545–564, 1999.

[18] P. Buchholz. Exact Performance Equivalence - An Equivalence Relation for Stochastic Automata. *Theoretical Computer Science 215(1/2)*, pages 239–261, 1999.

[19] P. Buchholz. Projection methods for the analysis of stochastic automata networks. In *Proc. 3rd int. workshop on the numerical solution of Markov chains, Prensas Universitarias de Zaragossa*, Zaragossa, Spain, 2000.

[20] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS Journal on Computing*, 12(3) :203–222, 2000.

[21] P. Buchholz, M. Fisher, and P. Kemper. Distributed steady state analysis using Kronecker algebra. In *Proc. 3rd int. workshop on the numerical solution of Markov chains, Prensas Universitarias de Zaragossa*, Zaragossa, Spain, 2000.

[22] P. Buchholz and P. Kemper. Efficient computation and representation of large reachability sets for composed automata. In *Proc. 5th workshop on Discrete Event Systems*, Ghent, Belgium, August 2000.

[23] J. Campos, M. Silva, and S. Donatelli. Structured solution of stochastic DSSP systems. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97), Saint Malo, France, IEEE Comp. Soc. Press*, pages 91 – 100, June 1997.

[24] R. Chan and W. Ching. Circulant preconditionners for stochastic automata networks. Technical report, Department of Mathematics, The Chinese University of Hong Kong, April 1998.

[25] G. Chiola, S. Donatelli, and G. Franceschinis. GSPN versus SPN : what is the actual role of immediate transitions ? In *Proc. 4th. int. conf. Petri Nets and Performance models*, Melbourne, Australia, December 1991.

[26] G. Chiola, G. Franceschini, R. Gaeta, and M. Ribaudo. GreatSPN 1.7 : GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24(1), 1996.

[27] G. Ciardo. Discrete-time Markovian stochastic Petri nets. In W. J. Stewart, editor, *Computation with Markov Chains*, pages 339–358. Raleigh, NC, USA, Jan 1995.

[28] G. Ciardo, R. German, and C. Lindemann. A Characterization of the Stochastic Process Underlying a Stochastic Petri Net. *IEEE Transactions on Software Engineering*, 20(7) :506–515, Jul 1994.

[29] G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. SMART : Stochastic Model Analyzer for Reliability and Timing. In *Tools of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 29–34, Sept. 2001.

[30] G. Ciardo and A.S. Miner. Storage Alternatives for Large Structured State Spaces. In R.A. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, St. Malo, France*, volume 1245 of *Lecture Notes in Computer Science*, pages 44–57. Springer-Verlag, June 1997.

[31] G. Ciardo and A.S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *In proc. 8th Workshop on Petri Nets and Performance models, IEEE CS Press*, Zaragossa, 1999.

[32] G. Ciardo and A.S. Miner. Efficient reachability set generation and storage using decision diagram. In *In proc. 20th int. Conf. Application and Theory of Petri Nets, LNCS 1639, Springer*, 1999.

[33] G. Ciardo and R. Zijal. Well-Defined Stochastic Petri Nets. In *Proc. 4th Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'96)*, pages 278–284. San Jose, CA, USA, 1996.

[34] M. Davio. Kronecker products and shuffle algebra. *IEEE Transactions on Computers*, 30(2) :116–125, 1981.

[35] T. Dayar and E. Uysal. Iterative methods based on splittings for stochastic automata networks. *European Journal of Operation Research*, 110(2) :166–186, 1998.

[36] D.D. Deavours and W.H. Sanders. "On-The-Fly" Solution Techniques for Stochastic Petri Nets and Extensions. In *Proceedings of the Seventh International Workshop on Petri Nets and Performance Models*, pages 132–141, Saint Malo, France, June 1997. IEEE Computer Society.

[37] D.D. Deavours and W.H. Sanders. An Efficient Disk-Based Tool for Solving Large Markov Models. *Performance Evaluation*, 33(1) :67–84, 1998.

[38] S. Donatelli. Superposed Stochastic Automata : a class of stochastic Petri nets with parallel solution and distributed state space. *Performance Evaluation*, 18(1) :21–36, 1993.

[39] S. Donatelli. Superposed Generalized Stochastic Petri nets : definition and efficient solution. In Valette, R., editor, *Lecture Notes in Computer Science ; Application and Theory of Petri Nets 1994, Proceedings 15th International Conference, Zaragoza, Spain*, volume 815, pages 258–277. Springer-Verlag, 1994.

[40] C. Durbach, F. Quessette, and A. Troubnikoff. Solving Large Markov Model based on Stochastic automata networks. In *Proc. Advances in Computer and Information Sciences'98, IOS Press*, Belek-Antalya, Turkey, 1998.

[41] P. Fernandes. *Méthodes Numériques pour la solution de systèmes Markoviens à grand espace d'états*. PhD thesis, Institut National Polytechnique de Grenoble, France, 1998.

[42] P. Fernandes, B. Plateau, and W.J. Stewart. Efficient Descriptor-Vector Multiplications in Stochastic Automata Networks. *JACM*, 45(3) :381–414, 1998.

[43] P. Fernandes, B. Plateau, and W.J. Stewart. Optimizing tensor product computations in Stochastic Automata Networks. *RAIRO, Operations Research*, 32(3) :325–351, 1998.

[44] J-M. Fourneau. Réseaux d'automates stochastiques : Exemples et Applications aux télécommunications. In *Proc. Ecole d'informatique des systèmes parallèles et répartis*, Toulouse, France, 1997.

[45] J-M. Fourneau. Stochastic Automata Networks : Using structural properties to reduce the state space. In *Proc. 3rd int. workshop on the numerical solution of Markov chains, Prensas Universitarias de Zaragossa*, Zaragossa, Spain, 2000.

[46] J-M. Fourneau, K-H. Lee, and B. Plateau. PEPS : A package for solving complex Markov models of parallel systems. In R. Puigjaner and D. Potier, editors, *Proceedings of the Fourth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 291–305, Palma, Spain, 1988.

[47] J-M. Fourneau and F. Quessette. Graphs and Stochastic automata networks. In *Proc. 2nd int. workshop on the numerical solution of Markov chains, Kluwer ed. Boston, Masachussetts*, 1995.

[48] H. Garavel and H. Hermanns. On Combining Functional Verification and Performance Evaluation using CADP. Technical report, Rapport de recherche INRIA n. 4492, France, July 2002.

[49] E. Gelenbe and I. Mitrani. *Analysis and synthesis of computer systems*. Academic Press (London and New York), 1980.

[50] E. Gelenbe and G. Pujolle. *Introduction to Queueing Networks, second edition*. John Wiley & Sons, 1998.

[51] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze. A general way to put time in petri nets. In *Proc. 5th Int. Workshop on Software Specification*, pages 60–67. Pittsburgh, Pennsylvania, USA, May 1989.

[52] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, New-York, 1994.

[53] J. Granata, M. Conner, and R. Tolimeri. Recursive fast algorithm and the role of the tensor product. *IEEE Transactions on Signal Processing*, 40(12) :2921–2930, Dec. 1992.

[54] O. Gusak and T. Dayar. Discrete-time stochastic automata networks and their efficient analysis. *Accepted for publication in Performance Evaluation*, 2002.

[55] O. Gusak, T. Dayar, and J-M. Fourneau. Stochastic Automata Networks and Near Complete Decomposability. *SIAM Journal on Matrix analysis and Applications*, 23 :581–599, 2001.

[56] O. Gusak, T. Dayar, and J-M. Fourneau. Lumpable continuous-time stochastic automata networks. *European Journal of Operational Research*, accepted for publication, 2001.

[57] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositionnal Performance Modelling with TIPtool. In *Proc. 10th Conf. Computer performance Evaluation : Modelling Techniques and Tools, Springer LNCS 1469*, Palma de Majorca, Spain, 1998.

[58] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagram to Represent and Analyse Continuous Time Markov Chains. In *Proc. 3rd int. workshop on the numerical solution of Markov chains, Prensas Universitarias de Zaragossa*, Zaragossa, Spain, 2000.

[59] J. Hillston. Compositional Markovian Modelling Using a Process Algebra. In W.J. Stewart, editor, *Numerical Solution of Markov Chains.*, Kluwer, 1995.

[60] J. Hillston and L. Kloul. An efficient Kronecker representation for PEPA models. In L. de Alfaro and S. Gilmore, editor, *Proceedings of the first joint PAPM-PROBMIV Workshop, Lecture Notes in Computer Science*, volume 2165, Aachen, Germany, Sept. 2001. Springer-Verlag.

[61] C. Hirel, B. Tuffin, and K. Trivedi. SPNP : Stochastic Petri Nets. Version 6.0. In *Proc. 11th Conf. Computer performance Evaluation : Modelling Techniques and Tools, Springer LNCS 1786*, Schaumburg, USA, March 2000.

[62] M.A. Holliday and M.K. Vernon. A Generalized Timed Petri Net Model for Performance Analysis. *IEEE Transactions on Software Engineering*, 13(12) :1297–1310, Dec 1987.

[63] J.-P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and Symbolic CTMC Model Checking. *Lecture Notes in Computer Science*, 2165, 2001.

[64] J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. Van Nostrand Reinhold, New York, 1960.

[65] J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. Springer New York, Heidelberg, Berlin, 1976.

[66] P. Kemper. Numerical Analysis of superposed GSPNs. *IEEE Trans. on Software Engineering*, 22(9), Sept. 1996.

[67] P. Kemper. Reachability analysis based on structured representations. In W. Reisig J. Billington, editor, *Proc. 17th int. conf. Application and Theory of Petri Nets, Springer LNCS 1091*, pages 269–288, 1996.

[68] P. Kemper. Transient Analysis of Superposed GSPNs. In *Proc. Petri Nets and Performance Models, PNPM'97*, Saint Malo, France, 1997.

[69] P. King and R. PooLey. Derivation of Petri Nets Performance Models from UML Specifications. In *Proc. 11th Conf. Computer performance Evaluation : Modelling Techniques and Tools, Springer LNCS 1786*, Schaumburg, USA, March 2000.

[70] P. Lascaux and R. Théodor. *Analyse numérique matricielle appliquée à l'art de l'ingénieur*. Masson, 1994.

[71] J. Ledoux. On weak lumpability of denumerable markov chains. Technical report, Rapport de recherche INRIA n. 2221, France, 1995.

[72] J. Ledoux. Weak lumpability of finite markov chains and positive invariance of cones. Technical report, Rapport de recherche INRIA n. 2801, France, 1996.

[73] M.A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets*. John Wiley & Sons, 1995.

[74] W. Massey. Open networks of queues : Their algebraic structure and estimating their transient behavior. *Advances in Applied Probability*, 16, no 1, march 1984.

[75] A.S. Miner. Efficient solution of GSPNs using Canonical Matrix Diagrams. In *Proc. PNPM'01, 9th International Workshop on Petri Nets and Performance Models*, pages 101–110, Aachen, Germany, Sept. 2001.

[76] A.S. Miner, G. Ciardo, and S. Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. In *Measurement and Modeling of Computer Systems*, pages 207–216, 2000.

[77] M.K. Molloy. Discrete Time Stochastic Petri Nets. *IEEE Transactions on Software Engineering*, 11(4) :417–423, Apr 1985.

[78] J. K. Muppala, G. Ciardo, and K. S. Trivedi. Modeling Using Stochastic Reward Nets. In *MASCOTS'93 International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, San Diego, CA*, pages 367–372. IEEE Comp. Soc. Press., Jan 1993.

[79] D.R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[80] B. Plateau. *De l'Evaluation du parallélisme et de la synchronisation*. PhD thesis, Université de Paris XII, Orsay (France), 1984.

[81] B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. In *Proc. ACM Sigmetrics Conference on Measurement and Modelling of Computer Systems*, Austin, Texas, Aug 1985.

[82] B. Plateau and K. Atif. Stochastic automata networks for modelling parallel systems. *IEEE Transactions on Software Engineering*, 17(10) :1093–1108, 1991.

[83] M. Rosenblatt. Functions of a Markov process that are Markovian. *J. of Math. and Mech.*, 8(4) :585–596, 1959.

[84] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, 1996.

[85] W. H. Sanders and J. F. Meyer. Reduced Base Model Construction Methods for Stochastic Activity Networks. *IEEE Journal on Selected Areas in Communications*, 9(1) :25–36, 1991.

[86] M. Siegle. Structured Markovian Performance Modelling with Automatic Symmetry Exploitation. In *Proc. 11th Conf. Computer performance Evaluation : Modelling Techniques and Tools, LNCS 794*, 1994.

[87] M. Siegle. Using structured modelling for efficient performance prediction of parallel systems. In *Parallel Computing : Trends and Applications (G.R.Joubert et al., eds.)*, pages 453–460, NorthHolland, 1994.

[88] R. Stansifer and D. Marinescu. Petri net models of concurrent Ada programs. *Microelectronics and Reliability*, 31(4) :577–594, 1991.

[89] W. J. Stewart. *An introduction to numerical solution of Markov chains*. Princeton University Press, New Jersey, 1994.

[90] W.J. Stewart. MARCA : Markov chain analyzer, a software package for Markov modelling. In W.J. Stewart, editor, *Numerical Solution of Markov Chains*, Marcel Dekker, 1991.

[91] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.

[92] C. Tadonki and B. Philippe. Parallel Multiplication of a Vector by a Kronecker product of Matrices. *Journal of Parallel and Distributed Computing and Practices*, 2(4), Dec. 1999.

[93] C. Tadonki and B. Philippe. Méthodologie de conception d'algorithmes efficaces pour le produit tensoriel. In *CARI 2000*, Madagascar, 2000.

[94] C. Tadonki and B. Philippe. Parallel Multiplication of a Vector by a Kronecker product of Matrices (Part II). *Journal of Parallel and Distributed Computing and Practices*, 3(3), Sept. 2000.

[95] F. Valois. *Modélisation et Evaluation de Performances de Réseaux Cellulaires Hiérarchiques*. PhD thesis, University of Versailles, France, 2000.

[96] V. Vèque and J. Ben-Othman. MRAP : A multiservices resource allocation policy for wireless ATM network. *Computer Networks and ISDN Systems*, 29(17–18) :2187–2200, 1998.

[97] R. Zijal and G. Ciardo. Discrete Deterministic and Stochastic Petri Nets. Technical report, ICASE Technical Report 96-72, Institute for Computer Applications in Science and Engineering, NASA/Langley Research Center, Hampton, VA, 1996.

[98] R. Zijal and R. German. A New Approach to Discrete Time Stochastic Petri Nets. In *Proc. 11th Int. Conf. on Analysis and Optimization of Systems, Discrete Event Systems*, pages 198–204. Sophia-Antipolis, France, Jun 1996.

[99] A. Zimmermann, J. Freiheit, and G. Hommel. Discrete Time Stochastic Petri Nets for Modeling and Evaluation of Real-Time Systems. In *Int. Workshop on Parallel and Distributed Real-Time Systems (invited paper)*. San Francisco, Apr 2001.

[100] W.M. Zuberek. Timed petri nets, definitions, properties, and applications. *Microelectronics and Reliability*, 31(4) :627–644, 1991.

Titre : Méthodes et algorithmes pour l'évaluation des performances des systèmes informatiques à grand espace d'états.

Résumé : Les chaînes de Markov facilitent l'analyse des performances des systèmes dynamiques dans de nombreux domaines d'application. Elles sont souvent utilisées par le biais de modèles de haut niveau. Parmi les différents formalismes couramment utilisés, on se place dans le cadre des réseaux d'automates stochastiques.
Cette thèse présente le formalisme des réseaux d'automates stochastiques (SANs) à temps continu, et propose un nouveau formalisme pour modéliser les systèmes à temps discret. Un des gros avantages des SANs réside dans la structure du générateur de la chaîne de Markov, qui est basée sur une formule tensorielle (et appelée *descripteur*).
Le principal objectif de ces travaux consiste à améliorer les méthodes et algorithmes existants sur le formalisme SAN, dans le but de pouvoir évaluer les performances de systèmes informatiques à grand espace d'états.
Pour cela, nous introduisons tout d'abord le concept de réseaux d'automates stochastiques avec réplication, et présentons des techniques d'agrégation exacte permettant de simplifier le modèle étudié en réduisant la taille de l'espace d'états.
L'évaluation de performance cherchée réside dans la détermination de l'état stationnaire de la chaîne de Markov. On utilise pour cela des méthodes itératives, et l'opération de base est la multiplication d'un vecteur par le descripteur du SAN. Les coûts de cette opération sont souvent très élevés lorsque l'espace d'états est grand, tant au niveau du temps d'exécution que de la place mémoire utilisée. Nous présentons des améliorations de l'algorithme classique en tenant compte du fait que dans de nombreux modèles, la proportion d'états accessibles est faible.
Les méthodes et algorithmes développés au cours de la thèse ont été implémentés dans le logiciel *PEPS 2003*. Des exemples numériques sont présentés pour illustrer les apports de cette thèse.

Mots clés : Évaluation des performances, Grand espace d'états, Réseaux d'automates stochastiques, Algèbre tensorielle, Temps continu et temps discret, Réplication d'automates, Agrégation exacte, Multiplication vecteur-descripteur, Algorithme du Shuffle.

Title : Methods and algorithms for the performance evaluation of systems with a large state space.

Abstract : Markov Chains facilitate the performance analysis of dynamic systems in many areas of application. This thesis presents the formalism of stochastic automata networks (SANs) to represent Markov systems.
The main goal of this work consists in improving existing methods for the performance evaluation of systems with a large state space. For this, we introduce the concept of SANs with replicas, and techniques to reduce the state space of such models.
To obtain performance indices, we propose an improvement of the basic operations by taking into account the fact that inside the product state space, the actual reachable state space can be much smaller. The new methods and algorithms have been implemented into the *PEPS 2003* software. Numerical examples are provided to illustrate the contributions of this thesis.

Keywords : Performance Evaluation, Large state space, Stochastic Automata Networks, Tensor algebra, Continuous and Discrete time, Automata replication, Exact agregation, Vector-descriptor product, Shuffle algorithm.