THÈSE

présentée à l'École Normale Supérieure de Cachan

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Cachan

par : Marie Duflot

Spécialité : Informatique

Algorithmes distribués sur des anneaux paramétrés

—

Preuves de convergence probabiliste et déterministe

Soutenue le 15 septembre 2003 devant un jury composé de :

– Joffroy Beauquier	rapporteur
– Ahmed Bouajjani	examinateur
– Laurent Fribourg	directeur de thèse
– Marta Kwiatkowska	examinatrice
– Catuscia Palamidessi	rapportrice
– Claudine Picaronny	examinatrice

Résumé

Cette thèse se situe dans le cadre de la vérification de systèmes distribués (machines organisées en réseau). Plus précisément, nous nous intéressons aux méthodes de preuve de convergence d'algorithmes distribués s'exécutant sur des réseaux en anneau de taille paramétrée. La convergence est une propriété assurant que l'algorithme atteint toujours (ou avec probabilité 1 dans le cadre probabiliste) un état légitime (c'est à dire satisfaisant une certaine propriété).

La difficulté principale est d'obtenir des preuves de convergence valides quelle que soit la taille du réseau et quel que soit l'ordre dans lequel sont activées les machines (ordonnancement). Nous proposons un cadre formel qui permet de simplifier des méthodes de preuves existantes.

Les principaux résultats sont les suivants.

- Dans le cadre des algorithmes probabilistes, nous avons exhibé un critère local de convergence permettant de démontrer la convergence d'algorithmes, quel que soit l'ordonnancement.
- En appliquant notre méthode à l'algorithme du dîner des philosophes probabilistes, nous avons montré que l'hypothèse d'équité sur les machines, considérée jusqu'ici comme nécessaire pour démontrer la convergence, est superflue. Nous avons de plus exprimé pour la première fois le temps moyen de convergence de cet algorithme en nombre d'actions.
- Dans le cas des algorithmes déterministes, nous avons amélioré l'automatisation d'une méthode de preuve de convergence fondée sur la réécriture (en utilisant une stratégie de réécriture préfixe).

  Nous avons ainsi démontré automatiquement la convergence de plusieurs algorithmes, tirés notamment du domaine de l'auto-stabilisation.

Remerciements

Je tiens tout d'abord à remercier Laurent Fribourg, pour m'avoir encadrée durant ces trois années de thèse, avec toute la disponibilité et la patience dont il a su faire preuve à mon égard. Il a su me guider et me motiver dans mon travail de recherche, tant dans les moments de correction intensive avant soumission que pendant des discussions scientifiques, parfois animées :o), que nous avons eu tout au long de ma thèse.

Je suis également reconnaissante envers tous les membres de mon jury qui ont contribué chacun à un moment de ma thèse, par des discussions éclairantes et parfois en m'apportant un point de vue différent sur des sujets proches de mes intérêts de recherche. Merci donc à Joffroy Beauquier, Ahmed Bouajjani, Marta Kwiatkowska et Catuscia Palamidessi. Un merci tout particulier à Claudine Picaronny, avec qui j'ai plus spécialement travaillé, et qui m'a également soutenue moralement lors des discussions ci-dessus citées.

J'ai profité pendant ces trois ans de l'environnement autant sympathique que scientifique du LSV, qui m'a permis de réaliser cette thèse dans les meilleures conditions, et cela grâce à tous ses membres, thésards et autres, qui font du LSV un lieu où il fait bon travailler, et se retrouver autour d'une pause café. Un merci particulier à François Laroussinie, qui a su m'encadrer sur le plan de l'enseignement, et répondre à toutes les questions que je pouvais me poser, tant matérielles, sur mon avenir, ou tout autre sujet qui me passait par la tête.

Sur un plan plus personnel, je tiens à remercier pas mal de cachanais, beaucoup ayant migré depuis, pour l'ambiance détendue et sympathique de ces trois années. Merci donc à Alex pour avoir eu le courage de partager mon bureau et les "quelques" bavardages qui allaient avec, à Pôti pour tellement de choses dont les soirées papotage jusqu'à pas d'heure, les corrections de devoir maison (si si, ça peut être rigolo), le défoulement sur ordinateur par jeu interposé, et j'en passe.

Merci aussi à mes parents, pour m'avoir permis d'en arriver là, pour leur soutien et "last but not least", pour la réalisation d'un délicieux pot de thèse !!!

Une pensée toute particulière va à Steve, pour sa présence à mes côtés (souvent à quelques centaines de kilomètres près :o) mais toujours dans un petit coin de mon écran d'ordinateur), pour avoir supporté mes sautes d'humeur (surtout en fin de rédaction), pour avoir relu quasiment toute ma thèse alors qu'il était en train de rédiger la sienne, et pour plein d'autres choses encore.

Et pour finir il ne faut pas oublier Vincent pour sa cravate, ainsi que tous ceux qui ont fait de ces trois années (et des autres) une période joyeuse et sportive, à savoir les handballeuses de Cachan (et les entraîneurs !!), les rugbywomen, les amateurs de jeux de sociétés, et tous les autres...

Table des matières

Résumé **3**

Remerciements **5**

Table des matières **7**

1 Introduction **11**

1.1 Enjeux de la vérification . . . 11
1.2 Méthodes formelles de vérification . . . 12
1.3 Propriétés à vérifier . . . 12
Propriétés de sûreté . . . 13
Propriétés de vivacité . . . 13
1.4 Systèmes distribués . . . 13
1.5 Aspect paramétré . . . 14
1.6 Systèmes probabilistes . . . 15
Intérêts des probabilités . . . 15
Modèles de systèmes probabilistes . . . 16
Vérification qualitative . . . 16
Vérification quantitative . . . 17
1.7 Contenu de la thèse . . . 17
Systèmes paramétrés probabilistes . . . 18
Systèmes paramétrés déterministes . . . 20
1.8 Plan de la thèse . . . 21

I Introduction aux probabilités et systèmes distribués **23**

2 Le formalisme probabiliste **25**

Non déterminisme *vs* “probabilisme” . . . 25
2.1 Quelques notions de probabilités . . . 25
2.2 Chaînes de Markov en temps discret . . . 28
2.2.1 Premières définitions . . . 29
2.2.2 Probabilités sur les chemins . . . 30
2.2.3 États récurrents, états transitoires . . . 31
2.2.4 Temps moyen de convergence . . . 33
2.3 Processus de décision markoviens . . . 34
2.3.1 Espaces de probabilités pour un processus de décision markovien . . . 35

Ordonnancement 35
Mesure de probabilités sur les exécutions 36
2.3.2 D'un processus de décision markovien à une chaîne de Markov 37

3 Systèmes distribués 39
3.1 Introduction 39
3.1.1 Notions de base 40
3.1.2 Objectifs des systèmes distribués 40
3.1.3 Problèmes soulevés par les systèmes distribués 41
3.1.4 Quelques questions étudiées en algorithmique distribuée 42
Élection 42
Consensus 42
Exclusion mutuelle 43
Allocation de ressources 43
Arbres couvrants 43
Détection de terminaison 44
Orientation 44
3.1.5 Tolérance aux fautes 45
3.2 Le modèle 46
3.2.1 Modes de communication 46
Communication par échange de messages 46
Communication par variables partagées 46
3.2.2 Topologie des systèmes distribués 47
3.2.3 Systèmes de transitions 48
3.2.4 Description des actions 48
3.3 Systèmes distribués en anneau 49
3.4 Systèmes distribués et réécriture 49
3.5 Démons 51
3.5.1 Notion de démon 51
3.5.2 Caractéristiques des démons 52
3.5.3 La propriété d'équité 53
3.6 Vérification de systèmes distribués 54
3.6.1 Logiques temporelles 55
3.6.2 Classification des propriétés 55
Sûreté 55
Vivacité 55
Caractérisation de ces propriétés 56
Méthodes classiques de vérification de propriétés de sûreté 56
3.6.3 Propriété de convergence 56
3.6.4 Auto-stabilisation 57
3.7 Systèmes paramétrés et indécidabilité 59
3.8 Systèmes distribués probabilistes 61
3.8.1 Logique temporelle probabiliste 63
3.8.2 Convergence probabiliste 64

II Convergence de systèmes paramétrés probabilistes 65

4 Critère formel de convergence 67
4.1 Algorithmes probabilistes vus comme des chaînes de Markov . . . 67
4.2 Cas d'un ordonnancement arbitraire . . . 70
4.2.1 Convergence de l'algorithme d'Israeli et Jalfon . . . 72
4.2.2 Convergence de l'algorithme de Beauquier, Gradinariu et Johnen . . . 73
4.2.3 Convergence de l'algorithme de Kakugawa et Yamashita . . . 76
4.3 φ-algorithmes . . . 79
4.4 Calcul du temps moyen de convergence par lumping . . . 80
4.4.1 Temps moyen de convergence de l'algorithme d'Herman . . . 83
4.4.2 Temps moyen de convergence de l'algorithme d'Israeli et Jalfon . . . 84

5 Application au dîner des philosophes 87
5.1 Description du problème . . . 87
5.2 L'algorithme de Lehmann et Rabin . . . 89
5.3 L'hypothèse d'équité . . . 91
5.4 Notre variante . . . 92
5.4.1 Suppression des règles invariantes . . . 92
5.4.2 Comparaison avec l'algorithme original . . . 92
5.5 La propriété de progrès . . . 94
5.6 Preuve de progrès sans équité . . . 95
5.6.1 Idées de la preuve . . . 95
5.6.2 Le théorème de convergence . . . 99
5.6.3 Jetons et composantes Δ_1, Δ_2, Δ_3 . . . 99
5.6.4 Anti-jetons et composantes Δ_4, Δ_5, Δ_6 . . . 101
5.6.5 Fonction Δ et preuve de progrès . . . 103
5.7 Temps moyen de convergence . . . 108
5.8 Le dîner des philosophes courtois . . . 110

6 Intérêt des algorithmes sans équité 111
6.1 Dîner des philosophes et π-calcul . . . 111
6.1.1 Différents types de π-calcul . . . 112
6.1.2 Encodage . . . 113
La notion de verrous . . . 113
Problèmes de topologie . . . 114
6.2 Autres domaines concernés . . . 115
6.2.1 Internet . . . 115
6.2.2 Tolérance aux pannes . . . 116

III Convergence de systèmes paramétrés déterministes 117

7 Preuve de convergence par surréductions 119
7.1 Réécriture et réductions closes . . . 119
7.2 Auto-stabilisation vers un ensemble clos . . . 123
7.3 Dérivations du premier ordre . . . 123

7.3.1 Unification . . . 124
7.3.2 Surréduction . . . 125
7.3.3 Chaînes de réduction minimales issues de $Top_{\mathcal{S}}$ . . . 126
7.4 Auto-stabilisation au premier ordre . . . 126
7.5 Retour aux φ-algorithmes . . . 128
7.6 Schémas . . . 128
7.7 L'outil de preuve Poulet . . . 130
7.8 Exemples . . . 131
7.8.1 L'algorithme de Beauquier-Debas . . . 131
7.8.2 L'algorithme de Ghosh . . . 132

8 Preuve automatisée de convergence par réécriture préfixe 135
8.1 Quelques résultats de réécriture préfixe . . . 135
8.2 Retour sur les surréductions . . . 138
8.2.1 Extension de règles . . . 139
8.2.2 Ensembles engendrés par réductions et surréductions . . . 140
8.2.3 Surréductions closes . . . 141
8.3 Langages réguliers inévitables et convergence . . . 142
8.3.1 Système équitable *vs* système nœtherien . . . 143
8.3.2 Retour sur les ensembles inévitables . . . 144
8.4 Condition suffisante de fermeture de ${\mathcal{N}_{\mathcal{S}}}^*$ . . . 146

9 Applications et résultats 149
9.1 Preuve d'auto-stabilisation . . . 149
9.1.1 L'algorithme de Beauquier-Debas . . . 150
9.1.2 L'algorithme de Ghosh . . . 150
9.2 Preuve de vivacité pour la détection de terminaison . . . 151
9.2.1 L'algorithme de Dijkstra-Feijen-van Gasteren . . . 152
9.3 Implantation et résultats . . . 154
9.3.1 Le programme . . . 154
9.3.2 Les résultats . . . 154
Résultat pour le système $\mathcal{BD}$ . . . 155
Résultats pour le système $\mathcal{G}$ . . . 155
Résultats pour le système $\mathcal{DFG}^{-1}$ . . . 156

10 Conclusion 159
Systèmes paramétrés probabilistes . . . 159
Systèmes paramétrés déterministes . . . 160
Perspectives . . . 160

Bibliographie 163

Index 170

Chapitre 1

Introduction

1.1 Enjeux de la vérification

Du fait que les systèmes informatiques sont présents partout, depuis les téléphones jusque dans les voitures, le besoin de vérifier ces systèmes s'est grandement accru. Si les ordinateurs et les programmes courants sont fournis sans aucune garantie de leur bon fonctionnement, cela ne peut être le cas pour tous les programmes.

Que ce soit pour des questions d'ordre économique (gros équipements, produits commercialisés en de très nombreux exemplaires,...), ou de sécurité (mettant en jeu des vies humaines, informations confidentielles,...), il peut être nécessaire, avant sa mise en service réelle, de s'assurer qu'un programme ou un système va fonctionner “correctement”.
On peut facilement concevoir qu'un utilisateur tolère de devoir relancer occasionnellement un programme qui a engendré une erreur, mais par contre il n'est pas acceptable par exemple qu'un numéro de carte bleue circule “en clair” sur le réseau.

Les exemples ne manquent pas pour illustrer le besoin impérieux de vérifier, à différents niveaux, les systèmes informatiques. Parmi eux, nous allons en présenter un, devenu maintenant classique pour justifier la nécessité de la vérification : le lancement raté de la première fusée Ariane V.

Le 4 juin 1996, 37 secondes après le décollage, la fusée Ariane V a quitté sa trajectoire prévue, finissant par se briser et exploser en vol.
Après enquête, le comité chargé de déterminer la cause de l'accident est arrivé aux conclusions suivantes : le problème venait d'un module qui calculait une valeur liée à la vitesse horizontale de la fusée. Le résultat obtenu dépassait une valeur maximale tolérée, induisant ainsi une erreur qui a entraîné la perte de contrôle de la fusée.
L'ironie de l'histoire est que d'une part ce module, repris du système pilotant la fusée Ariane IV, était adapté pour la trajectoire d'Ariane IV, dont la composante horizontale de la vitesse était plus petite et donc pas adaptée au nouveau lanceur, et d'autre part, même dans Ariane IV, ce module n'était utile que dans les instants avant le décollage, et en particulier pas au moment où l'erreur s'est produite.
C'est donc un module non adapté au lanceur sur lequel il a été installé, et inutile au moment de l'erreur qui a provoqué l'explosion en vol du lanceur Ariane V, et les pertes financières qui en ont résulté. Pour plus de précisions, on peut trouver sur Internet le rapport d'accident en

anglais[1].

Pour une présentation détaillée d'exemples de défaillances informatiques et de leurs conséquences, parfois en vies humaines, on peut se référer à l'introduction de la thèse [Fle02].

Les nombreux exemples illustrent le fait que le succès d'un système informatique dépend de nombreux facteurs, allant de la réalisation d'un cahier des charges précis et correct à la formation des utilisateurs, en passant par des phases de vérification de cohérence entre les différents modules, voire un test avant l'utilisation réelle. À cela s'ajoutent des méthodes de vérification dite *formelle*, décrites dans la section suivante. La mise en oeuvre d'un système correct est donc un processus très complexe qui nécessite des vérifications à de nombreux niveaux ; un problème dans un seul des maillons de la réalisation pouvant causer l'échec de tout le projet.

1.2 Méthodes formelles de vérification

Parmi tous les facteurs permettant d'essayer de garantir le bon fonctionnement des systèmes, nous allons nous intéresser plus précisément à la vérification formelle, qui s'opère sur un modèle du système.
Comme cette vérification est une étape qui prend en général du temps, elle n'est pas réalisée de manière systématique sur tous les systèmes informatisés, mais sur ceux pour lesquels une erreur serait critique.
Comme les systèmes dits critiques peuvent être de natures très différentes, la vérification formelle est présente dans de très nombreux domaines informatiques, comme les protocoles cryptographiques, les systèmes temps-réel, les systèmes probabilistes, etc.

Il existe principalement trois approches dans la vérification formelle de logiciels, qui sont le model-checking (méthodes qui consistent à vérifier automatiquement qu'un modèle $\mathcal{S}$ du système satisfait une propriété φ, noté $\mathcal{S} \models \varphi$), la preuve assistée (utilisation de règles de déduction afin de prouver "mathématiquement" des propriétés), et le test (génération d'exécutions particulières afin de se convaincre qu'un système vérifie une propriété).

Cette thèse s'intéresse à la vérification de systèmes paramétrés, et utilise des méthodes automatiques relevant du model-checking.

On peut trouver des références sur les méthodes de test dans [BT01]. L'article [Rus01] présente une introduction sur la preuve assistée. On peut également citer quelques outils comme Isabelle/HOL [NPW02], COQ [BC] ou PVS [COR$^+$95]. En ce qui concerne le model-checking, on peut consulter les ouvrages [BBF$^+$01, SBB$^+$99] et [CGP99] pour une présentation des techniques et outils.

1.3 Propriétés à vérifier

Dans le cadre de la vérification, on distingue deux types de propriétés particulières, qui sont d'une part les propriétés de sûreté, et d'autre part les propriétés de vivacité. Ces propriétés peuvent porter soit sur les états du système, soit sur les chemins (c'est-à-dire les suites de transitions).

[1] disponible à l'adresse `http ://sunnyday.mit.edu/accidents/Ariane5accidentreport.html`

Propriétés de sûreté

Les propriétés de sûreté sont en général étudiées dans le cas des systèmes critiques, pour lesquels on veut détecter si le système ne va pas avoir un comportement aberrant, dangereux tant pour le système lui-même que pour les utilisateurs. On veut alors vérifier des propriétés du genre :

"Pour tout comportement du système, on n'atteint jamais un état d'erreur"

"Si à un moment le système est dans l'état alarme, alors il y a eu une panne auparavant"

La méthode de vérification consiste alors à partir de l'ensemble initial, calculer tous les états accessibles, et vérifier qu'aucun d'entre eux ne correspond à un comportement aberrant.

Propriétés de vivacité

La vivacité consiste à avoir le point de vue dual : on ne veut pas empêcher que quelque chose de mauvais arrive, mais plutôt s'assurer qu'un événement souhaité va finir par arriver. Un exemple classique de propriété de vivacité est la *convergence*, qui affirme que, quel que soit l'état de départ, quel que soit le chemin emprunté, on va arriver en un temps fini dans un ensemble d'états souhaité. En d'autres termes, on veut montrer que :

"Tous les chemins mènent à Rome."

Ce sont des propriétés de ce type que nous allons vérifier dans cette thèse.
Un cas particulier de propriété de convergence est l'*auto-stabilisation*, introduite par Dijkstra en 1974 [Dij74]. Dans ce cas on veut montrer que tous les chemins mènent à un certain ensemble dit *légitime*, et qu'ensuite le système à un comportement correct. Ce genre de propriétés est beaucoup étudié, notamment pour son intérêt dans le cadre de la tolérance aux fautes. En effet, un algorithme auto-stabilisant permet, après une panne momentanée, de revenir dans un ensemble d'états *légitimes* en un temps fini et en l'absence de nouvelles pannes. Le livre [Dol00] se consacre uniquement à l'auto-stabilisation.

1.4 Systèmes distribués

La notion de *système distribué*, ou *réparti* englobe tous les systèmes composés de plusieurs entités autonomes (processus, ordinateurs, etc.) communiquant chacune avec ses "voisins" au moyen soit de canaux de communication soit de variables partagées. Chaque entité évolue alors en fonction de son état propre et de celui de ses voisins. Ainsi les réseaux comme Internet entrent dans le cadre des systèmes distribués.

L'enjeu des algorithmes distribués est de réussir à faire collaborer toutes ces entités pour accomplir une tâche globale. Pour ne donner qu'un exemple, on peut citer le consensus, qui consiste à mettre d'accord tous les composants du système, ou du moins ceux qui ne sont pas en panne, sur une certaine valeur.

Les systèmes distribués vont nécessiter des techniques particulières lors de leur vérification. Les raisons en sont nombreuses, mais nous allons en citer deux à titre d'exemple.

Tout d'abord, du fait que les entités ne communiquent qu'avec un voisinage restreint, aucun des participants à l'algorithme n'a une connaissance globale de l'état du système (appelé *configuration*), ce qui accroît la difficulté de la vérification. De plus, on n'a pas forcément d'information sur l'ordre dans lequel les différentes entités vont effectuer des actions.

Formellement, pour représenter le fait que ce choix de l'ordre des processus n'est pas toujours prédéfini, on introduit la notion de *démon*, qui est un mécanisme extérieur, choisissant à chaque étape la ou les entités qui vont effectuer une action. Le démon est obligé de choisir des processus pouvant effectivement faire une action, mais parmi eux son choix peut être arbitraire. Les démons symbolisent donc la part de non-déterminisme dans les systèmes distribués.
Il est possible de faire des hypothèses sur le démon, comme par exemple l'équité qui consiste à supposer que, le long de toute exécution infinie, le démon ne peut pas indéfiniment ignorer un processus qui peut effectuer une action. Cette hypothèse est souvent utilisée et parfois on vérifie des algorithmes uniquement sous l'hypothèse que le démon est équitable. Cependant, dans le cas où l'on considère que les processus peuvent tomber en panne, il est déraisonnable de supposer que les processus vont effectuer infiniment souvent des actions.

Les ouvrages [Tel91, Tel94, Lyn96] donnent une bonne vue d'ensemble sur l'étude des algorithmes distribués.

Un problème classique en algorithmique distribuée est l'allocation de ressources. Plusieurs machines disposées en réseau partagent une ou plusieurs ressources, et on veut s'assurer qu'une ressource n'est pas utilisée simultanément par deux processus à la fois. Dans le cas où une seule ressource est partagée par tous les processus, on parle d'exclusion mutuelle. Un exemple très étudié d'algorithme d'allocation de ressources plus général est le dîner des philosophes ([Dij72]).

Nous allons dans cette thèse nous intéresser plus précisément à la vérification de systèmes distribués modélisés sous forme de N machines (N automates finis) disposées en anneau ou de manière linéaire.

1.5 Aspect paramétré

Lorsqu'un système à vérifier est fini (c'est à dire quand l'ensemble des configurations et l'ensemble des transitions possibles de ce système sont finis), les propriétés à vérifier sont en général décidables. Seulement, dès que l'on introduit une source d'infini dans le système, la vérification devient beaucoup plus délicate.

La source d'infini que nous allons considérer est l'introduction d'un paramètre. En effet, considérons un système composé de N machines, où N est un paramètre. Si on prend une valeur précise pour N, on a un système fini sur lequel on peut faire de la vérification classique. Si par contre on ne veut pas fixer la valeur de N, on se retrouve face à une collection infinie de systèmes finis.
Il se trouve que la vérification sur de tels systèmes, à savoir montrer qu'une propriété est vraie quel que soit le nombre N de processus dans le système est indécidable en général (voir [AK86]). C'est pourtant le problème auquel nous allons nous intéresser.

Du fait du résultat d'indécidabilité, on ne peut pas trouver d'algorithmes qui fonctionnent dans tous les cas. On peut alors avoir deux approches pour contourner ce problème :
- soit on développe des méthodes particulières qui portent sur des sous-classes décidables,
- soit on développe des méthodes qui ne sont pas garanties de terminer ou de donner un résultat exploitable.

Citons quelques approches utilisées pour vérifier des systèmes paramétrés.

Accélération : Au lieu d'appliquer une seule transition à chaque étape de vérification, l'accélération consiste à appliquer plusieurs transitions successives à la fois, et parfois même un nombre non borné de fois la même transition. Par exemple, pour calculer un ensemble d'états accessibles, on peut essayer de calculer en une seule étape l'ensemble des états accessibles en appliquant un nombre indéfini de fois une même transition. Cela permet de calculer plus d'états accessibles d'un coup, et ainsi d'arriver plus rapidement à l'ensemble de tous les états accessibles.

L'accélération ne permet pas de s'assurer que l'on atteindra toujours en temps fini l'ensemble souhaité, mais cette technique permet sur de nombreux exemples de le calculer efficacement.

La plupart des travaux utilisant des techniques d'accélération, que ce soit dans le cadre des systèmes paramétrés ou d'autres systèmes d'états infinis, servent à prouver des propriétés de sûreté par un calcul d'états accessibles ([FO97, BF99, JN00, BJNT00, Tou01, FL02]), parfois avec un critère permettant de s'assurer que le calcul va terminer ([BMT01]). Certaines méthodes ont fait l'objet d'outils, comme FAST[2] ou TReX[3]. D'autres travaux s'intéressent également à des propriétés de vivacité, comme [PS00], ou [BBFM01] qui utilise des techniques de réécriture.

Abstraction : Les techniques d'abstraction consistent à réduire le nombre d'états du système. Pour cela, à partir d'un système $\mathcal{S}$, on construit un nouveau système $\mathcal{S}_a$, appelé système abstrait. De même pour la propriété φ, on définit une propriété φ_a satisfaisant la condition suivante :

$$S_a \models \varphi_a \implies S \models \varphi$$

Une abstraction vérifiant une telle propriété est dite correcte. Le gros du travail est bien sûr de trouver une fonction d'abstraction correcte.

Pour une présentation de l'abstraction basée sur un cas d'étude, on peut regarder l'article [Val01]. Parmi les nombreux travaux basés sur l'abstraction dans le cadre des systèmes distribués on peut citer [EN95, CGJ95]. L'article [KM95] étudie une méthode d'abstraction utilisant des "invariants de réseau". Cette méthode est utilisée notamment dans [KPSZ02, APZ03] pour prouver une propriété de convergence probabiliste d'un algorithme d'allocation de ressources. Cette technique peut également être utilisée pour abstraire le nombre de participants dans un protocole cryptographique ([CLC03]).

Cette thèse va utiliser des méthodes relevant de l'accélération, notamment dans la partie de vérification des systèmes déterministes, où l'on va calculer automatiquement tous les états accessibles pour une relation de transition particulière.

1.6 Systèmes probabilistes

Intérêts des probabilités

Dans certains systèmes, on est amené à introduire la notion de probabilités, et ce pour plusieurs raisons.

Elle permet tout d'abord de modéliser de façon probabiliste certains comportements réels après observation statistique, comme par exemple sur des canaux non fiables, la probabilité qu'un message soit perdu (voir [BS03]).

[2]voir la page `http ://www.lsv.ens-cachan.fr/~leroux/fast/`

[3]voir la page `http ://www.liafa.jussieu.fr/~sighirea/trex/`

Les probabilités peuvent également être introduites pour des raisons d'efficacité . Dans certains cas, il est beaucoup plus rapide de montrer qu'un système satisfait une propriété avec une probabilité arbitrairement proche de 1, que de montrer qu'il la satisfait tout court. On applique ainsi une méthode de vérification probabiliste à un système qui ne l'est pas forcément, comme par exemple dans [LLM+02].
L'ajout de probabilités est même parfois nécessaire. En effet, il existe des problèmes pour lesquels il a été prouvé qu'il n'existe pas de solution déterministe. C'est souvent le cas dans des systèmes symétriques où les probabilités sont requises pour rompre la symétrie. Un exemple bien connu est celui du dîner des philosophes [Dij72], pour lequel Lehmann et Rabin ont montré qu'il n'existe pas de solution déterministe, symétrique et totalement distribuée. Ils donnent alors une solution probabiliste à ce problème [LR81, RL94].

Modèles de systèmes probabilistes

Dans la plupart des systèmes réels, il existe une part de non-déterminisme. Celui-ci peut représenter différentes choses, comme l'interaction avec l'environnement que l'on ne contrôle pas, une partie de la spécification du système qui n'est pas parfaitement connue, etc. Dans notre cadre, ce non déterminisme sera représenté par un démon, défini à la section 1.4

Pour représenter de tels systèmes, il faut alors un modèle qui comprenne à la fois du non-déterminisme et des probabilités. C'est le cas des processus de décision markoviens (voir par exemple [Put94], ou dans le cadre informatique [dA97]). Il existe des modèles similaires, comme par exemple les automates probabilistes [Rab63, SL95, Seg95], ou les chaînes de Markov concurrentes [Var85].

Sur de tels modèles, afin de définir des probabilités sur les chemins (ou exécutions), il faut fixer les choix que le démon va faire à chaque étape. Ceci est fait au moyen de ce que l'on appelle un ordonnancement qui, étant donné une exécution finie, renvoie une mesure de probabilité sur les différents choix possibles. On a alors, étant donné un état de départ s et un ordonnancement $\mathcal{O}$, un espace de probabilités sur les exécutions issues de s et suivant l'ordonnancement $\mathcal{O}$ (voir chapitre 2 pour des définitions précises).

Vérification qualitative

Dans le domaine probabiliste, on appelle vérification qualitative le fait de prouver que des propriétés sont satisfaites avec probabilité 0 ou 1. On s'intéresse donc uniquement à des propriétés du type :

"Tous les chemins, sauf un ensemble de probabilité 0, satisfont la propriété ϕ"

Dans ce cas, la valeur exacte des probabilités des transitions importe peu. Ce qui compte c'est de savoir si cette probabilité est nulle, égale à 1 ou strictement incluse entre ces deux valeurs.
Les probabilités servent donc à s'assurer que, lorsqu'une transition (probabiliste) a plusieurs résultats possibles, et si cette transition est effectuée une infinité de fois le long d'un chemin, alors elle ne va pas (ou uniquement avec probabilité 0) donner toujours le même résultat. Dans ce cas, les probabilités peuvent parfois être remplacées par ces conditions d'équité (voir par exemple [KPSZ02]).

Dans cette thèse, nous allons nous intéresser à des propriétés de convergence probabiliste, c'est à dire du genre :

"Tous les chemins, sauf un ensemble de probabilité 0*, mènent à Rome."*

Sur ce genre de système, si par exemple on montre une propriété de convergence avec probabilité 1 vers un certain ensemble, on peut alors s'intéresser au temps moyen pour atteindre cet ensemble. Pour de exemples de travaux sur ce calcul de temps moyen de convergence, on peut regarder [LSS94, PS95] pour l'algorithme du dîner des philosophes probabilistes. On peut aussi consulter [DIM95] pour une approche orientée théorie des jeux, qui permet de donner une borne supérieure au temps moyen de convergence.

Vérification quantitative

Par opposition à la méthode précédente, la vérification quantitative consiste pour sa part à s'intéresser précisément aux valeurs des probabilités, et prouver des propriétés du genre :

"Avec une probabilité supérieure à $0,9$*, le système va atteindre l'état s."*

Comme dans ces systèmes il faut conserver la valeur exacte des probabilités, cela rend la vérification plus difficile. Cependant, dans le cas où le système est fini, il existe des outils permettant de vérifier automatiquement des propriétés de logique temporelle probabiliste sur des systèmes. Parmi eux, nous pouvons citer l'outil PRISM ([KNP02]), qui a permis à ses concepteurs de prouver de nombreux exemples, notamment dans le domaine des systèmes distribués [4].

1.7 Contenu de la thèse

Dans cette thèse, nous nous intéressons au problème de vérifier des systèmes distribués probabilistes et déterministes. Plus précisément, nous nous intéressons à des propriétés de convergence de ces systèmes, c'est à dire :

"Quelle que soit la configuration de départ, on va arriver en un temps fini dans l'ensemble voulu."

Ici le temps est compté en nombre de transitions. Cela signifie que, quelle que soit la configuration initiale du système, quel que soit le chemin suivi (suivant l'algorithme), on va arriver au bout d'un nombre de transitions fini dans l'ensemble souhaité.

Ces propriétés de convergence servent à assurer qu'un bon événement (l'arrivée dans l'ensemble souhaité) va inéluctablement se produire, et permettent de vérifier qu'un système va effectivement réaliser la tâche voulue. La motivation d'étudier de telles propriétés nous est initialement venue pour étudier des systèmes auto-stabilisants. Comme on l'a vu précédemment, ce problème a été introduit par Dijkstra en 1974 [Dij74], et suscite depuis un grand intérêt. Ces systèmes auto-stabilisants ont en effet la propriété d'être tolérants aux fautes transitoires : si à un moment (suite à une défaillance passagère, une réinitialisation etc.) le système n'est plus dans un état correct, alors il va pouvoir, sans aide extérieure, revenir en un temps fini dans un ensemble d'états corrects, appelé ensemble *légitime*.

Sur des systèmes finis, vérifier la convergence est décidable. Seulement on ne va pas s'intéresser à des systèmes distribués finis, mais paramétrés. L'intérêt d'étudier de tels systèmes est

[4] l'outil à télécharger, ainsi qu'une série d'exemples commentés sont disponibles sur la page `http://www.cs.bham.ac.uk/~dxp/prism/`

le suivant : les méthodes classiques comme le model-checking ont une complexité qui dépend de la taille du système étudié, et donc plus le système est gros, plus il va devenir difficile à vérifier. Ici, comme on veut vérifier une propriété quel que soit le nombre N de processus dans le système, on est obligé de trouver une méthode qui fonctionne pour tout N, et donc en particulier quand le nombre de processus devient très grand.

Plus précisément, les systèmes que nous allons considérer auront une topologie soit en anneau (des processus disposés en cercle avec une communication entre un processus et ses deux voisins autour de la table), soit linéaire (une chaîne de processus, autrement dit un anneau pour lequel on a coupé la communication entre deux processus donnés).

D'autre part nous avons aussi travaillé sur des systèmes où la symétrie du problème empêche l'existence d'une solution déterministe. Nous nous sommes donc intéressés à des algorithmes distribués probabilistes, avec tout le formalisme et les méthodes de vérification spécifiques que ces algorithmes nécessitent.

Systèmes paramétrés probabilistes

Dans la partie probabiliste de notre travail, nous nous intéressons à prouver la convergence avec probabilité 1 d'algorithmes vers un ensemble légitime.

Pour cela il a d'abord fallu présenter formellement le cadre mathématique dans lequel ces systèmes s'inscrivent, en particulier les deux modèles sous-jacents : chaînes de Markov et processus de décision markoviens. Comme annoncé précédemment, les processus de décision markoviens permettent de représenter des systèmes mêlant à la fois non déterminisme et probabilités, ce qui s'adapte parfaitement au cadre des systèmes distribués où le choix non-déterministe est fait par un mécanisme extérieur appelé démon.

La vérification de propriétés de convergence sur de tels systèmes est en pratique souvent délicate, et elle l'est d'autant plus lorsque le démon peut choisir arbitrairement les processus qui vont réagir. Une méthode employée pour prouver la convergence est de montrer que l'on va inéluctablement (et avec probabilité non nulle) se rapprocher de l'ensemble voulu.
L'article de Beauquier, Durand-Lose, Gradinariu et Johnen ([BDLGJ02]) énonce un théorème (initialement présenté dans [BGJ99b]) qui utilise une telle méthode. Informellement, le résultat est le suivant : étant donnés un ordonnancement $\mathcal{O}$ et un ensemble de départ E_1, si, quel que soit l'état de E_1 dont on part, on arrive en un temps borné et avec probabilité non nulle dans un ensemble donné E_2 alors, pour cet ordonnancement $\mathcal{O}$, il y a convergence probabiliste vers E_2 en partant de E_1.

Dans notre travail, nous exhibons un critère local de convergence : nous étudions des systèmes munis d'une fonction (dépendante de l'algorithme) telle, que lorsqu'on n'est pas dans un certain ensemble $\mathcal{L}$, pour tout choix par le démon des processus qui vont effectuer une action, alors soit on arrive dans l'ensemble $\mathcal{L}$ soit la fonction décroît avec probabilité non nulle.
Notre résultat nous permet alors de prouver que, si une telle fonction existe, l'algorithme converge avec probabilité 1 quel que soit l'ordonnancement.

L'intérêt de notre méthode, qui peut être vue comme une restriction du théorème présenté dans [BDLGJ02] est justement son caractère local. Comme nous considérons de la décroissance en une étape, une fois que nous avons trouvé notre fonction, il suffit d'effectuer une étape de transition et de vérifier comment se comporte la fonction. Notre critère est donc plus restreint mais plus simple a vérifier en pratique sur des exemples, dès qu'une telle fonction est connue. De plus cette fonction nous permet d'abstraire le système. Lorsque les états ayant la même valeur (pour la fonction) ont également le “même comportement probabiliste”, on peut les regrouper sous forme de classes d'équivalence, et appliquer une méthode mathématique appelée *lumping* pour faciliter le calcul du temps moyen de convergence. Le calcul se fait alors sur l'ensemble des états quotienté par la relation d'équivalence.

Nous avons appliqué avec succès cette méthode, pour prouver la convergence d'algorithmes tirés de la littérature, dans le cas où le système se comporte comme une chaîne de Markov (Herman [Her90]), comme dans le cas où l'ordonnancement est arbitraire (Israeli et Jalfon ([IJ90]), Beauquier Gradinariu et Johnen ([BGJ99a]), et Kakugawa et Yamashita ([KY97])). Nous avons également donné des bornes supérieures pour le temps moyen de convergence des algorithmes de Herman et Israeli-Jalfon dans le cas où le système possède deux *jetons*.

Notre dernière contribution dans ce cadre probabiliste a été d'appliquer la méthode de preuve de convergence décrite précédemment à une variante sans équité de l'algorithme du dîner des philosophes probabilistes [LR81].

Cet algorithme bien connu permet de résoudre un problème d'allocation de ressource insoluble dans le cadre déterministe. Informellement, N philosophes sont assis autour d'une table, une baguette étant disposée entre deux philosophes “voisins”. Pour manger, un philosophe a besoin de tenir simultanément ses deux baguettes, à savoir celle partagée avec son voisin de gauche et celle partagée avec son voisin de droite. Le but de l'algorithme étant d'assurer que si un philosophe a faim, alors en un temps fini un philosophe (pas nécessairement le même) va manger.

Nous sommes partis de l'algorithme du dîner des philosophes probabilistes de Lehmann et Rabin [LR81] et avons essayé de prouver sa convergence sans hypothèse d'équité sur le démon. L'algorithme original suppose pour sa part que toute exécution possible va être équitable. Cette hypothèse d'équité permet d'introduire la notion de *tour*, c'est à dire une période de temps pendant laquelle chaque philosophe a effectué au moins une action. Pour pouvoir supprimer l'hypothèse d'équité, nous avons dû modifier quelque peu l'algorithme, en enlevant les boucles (ou transitions invariantes) qui ne modifiaient pas la configuration du système. En effet, si on autorise un démon a choisir toujours le même processus qui peut faire une transition invariante, alors le système peut rester indéfiniment dans la même configuration et ne pas converger.

Nous avons ensuite exhibé une fonction Δ à sept composantes qui décroît avec probabilité non nulle pour l'ordre lexicographique, à chaque application de transition, sauf lorsqu'un philosophe mange. Ceci nous a permis de prouver la convergence probabiliste de l'algorithme vers l'ensemble des configurations où au moins un philosophe est en train de manger.

Cette variante sans hypothèse d'équité nous a permis d'aborder pour la première fois le temps moyen de convergence en terme de nombre de transitions, et non plus de nombre de tours comme le faisaient les approches précédentes. Nous avons donc montré que pour un certain démon malicieux, ce temps moyen de convergence peut être au moins exponentiel en le nombre de philosophes. Cette complexité n'était pas apparente dans les travaux précédents, car si on compte le temps en nombre de tours, alors on a un temps moyen de convergence qui est constant.

D'autre part, on peut mentionner que le fait de supprimer l'hypothèse d'équité dans cet algorithme est justifié. En effet, cette variante du dîner des philosophes probabilistes a été utilisée indépendamment par Herescu et Palamidessi pour résoudre un problème d'allocation de ressource afin de réaliser l'encodage du π-calcul synchrone dans le π-calcul probabiliste asynchrone (voir l'article [PH02]). La question de considérer des algorithmes sans hypothèse d'équité se pose également dans des domaines comme Internet. D'une part certains participants à l'algorithme peuvent tomber en panne, et d'autre part on peut vouloir donner un service moins équitable, à savoir servir en priorité certains participants, en fonction du prix que ceux-ci ont payé pour le service.

Systèmes paramétrés déterministes

Dans le cadre déterministe, nous avons abordé le problème sous un angle différent. Nous sommes partis d'un travail de Beauquier, Bérard, Fribourg et Magniette publié dans [BBFM01] permettant de prouver la convergence d'algorithmes sur des anneaux paramétrés, et ce en utilisant des techniques de réécriture, et plus particulièrement les surréductions. Nous avons ensuite effectué un pas supplémentaire vers l'automatisation de cette méthode.

Le travail de [BBFM01] permet, en restreignant les exécutions à considérer, d'essayer de construire un graphe de transitions fini représentant le système, et tel que si sur ce graphe toutes les exécutions convergent vers l'ensemble voulu, alors l'algorithme original va lui aussi converger.
Le principe est tout d'abord de ne considérer que des systèmes pour lesquels toute exécution infinie va contenir une infinité d'applications de règles à la position la plus à droite (appelée *top*), puis de ne considérer que des exécutions qui commencent par l'application d'une telle règle. Ensuite en utilisant des variables du premier ordre permettant de représenter symboliquement plusieurs processus consécutifs, on va partir non plus d'un mot sans variable mais d'un mot avec variable correspondant au membre droit de la règle appliquée en top et "deviner" petit à petit, grâce aux surréductions, les lettres qui étaient représentées par la variable. Enfin, en regroupant de tels mots avec variables sous forme de *schémas*, il est possible de diminuer grandement la taille du graphe de transitions représentant l'algorithme, et éventuellement obtenir un graphe fini, qui est une sur-approximation du graphe de transitions original. La dernière étape consiste alors à prouver qu'il n'y a pas de "mauvais" cycles sur ce graphe, c'est à dire ne passant pas par l'ensemble légitime $\mathcal{L}$.

A partir de ces résultats, nous avons utilisé la restriction des exécutions grâce aux variables du premier ordre et en ne considérant que des exécutions commençant par l'application d'une règle en position top.
Nous avons ensuite montré en quoi les étapes de surréduction pouvaient être envisagées comme de la réécriture préfixe (modifiant les premières lettres du mot) ou suffixe (modifiant les dernières lettres du mot). Ceci nous a permis d'utiliser des résultats de réécriture préfixe dûs à Caucal [Cau90] permettant de calculer automatiquement et en temps polynomial le langage engendré par réécriture préfixe itérée.
Nous avons ensuite exhibé un critère sous lequel le langage $\mathcal{N_S}^*$ engendré par réécriture préfixe et suffixe itérées est fermé pour le système de réécriture considéré. Ceci nous permet, lorsque le système est équitable (*i.e.* tout démon possible pour ce système est forcément équitable), d'exhiber un ensemble inévitable (c'est à dire par lequel toute configuration infinie va passer) et régulier.
Il nous reste alors à tester que l'ensemble inévitable calculé est inclus dans $\mathcal{L}$ pour en déduire que $\mathcal{L}$ est inévitable.
Nous avons également donné un critère, celui de système *unidirectionnel*, vérifiable directement sur les règles, permettant d'assurer que $\mathcal{N_S}^*$ est fermé, et donc appliquer notre méthode.

Enfin nous avons réalisé un programme en Prolog implantant l'algorithme de Caucal dans ce contexte, et permettant ainsi de calculer automatiquement les ensembles inévitables. Ce programme a été utilisé pour vérifier plusieurs exemples, tirés du domaine de l'auto-stabilisation et de la détection de terminaison.

1.8 Plan de la thèse

Cette thèse est découpée en trois parties. Tout d'abord la partie "Introduction aux probabilités et systèmes distribués" introduit les notions qui seront nécessaires dans la suite de cette thèse. Le **chapitre 2** donne une introduction sur les notions de probabilités et des modèles stochastiques permettant de représenter les systèmes probabilistes que nous considérons à la partie II. Ensuite le **chapitre 3** introduit les systèmes distribués, et présente une partie de la théorie qui y est rattachée. C'est aussi dans ce chapitre que sont présentées les propriétés que l'on va par la suite vouloir vérifier sur ces systèmes.

La partie "Convergence de systèmes paramétrés probabilistes" contient tout le travail effectué lors de cette thèse concernant les systèmes distribués probabilistes. Le **chapitre 4** présente la théorie, et principalement un critère permettant de s'assurer de la convergence du système étudié. Il présente également une façon, basée sur une méthode dite de *lumping*, pour calculer plus efficacement des temps moyens de convergence pour les systèmes se comportant comme des chaînes de Markov. Le **chapitre 5**, présente pour sa part une application majeure des résultats du chapitre précédent : l'étude et la preuve de convergence d'une variante sans équité de l'algorithme du dîner des philosophes probabilistes de Lehmann-Rabin. Enfin le **chapitre 6** argumente l'intérêt d'étudier des systèmes sans équité, comme par exemple l'utilisation de la même variante de l'algorithme du dîner des philosophes pour l'encodage du π-calcul dans une variante probabiliste et asynchrone.

La partie "Convergence de systèmes paramétrés déterministes" décrit enfin le travail effectué dans le cadre non probabiliste. Tout d'abord le **chapitre 7** présente un travail effectué par Beauquier, Bérard, Fribourg et Magniette à propos de preuves de convergence de systèmes auto-stabilisants. En utilisant un résultat de réécriture dû à Dershowitz, les auteurs considèrent un sous-ensemble des exécutions possibles et réussissent, sur ce sous-ensemble, à prouver que toute exécution atteint l'ensemble *légitime* souhaité. Le **chapitre 8** part des résultats du chapitre précédent pour automatiser encore un peu plus cette méthode de preuve de convergence. En se basant sur un résultat de Caucal, il est même possible de construire en temps polynomial un automate reconnaissant un certain langage *inévitable*[5] et ainsi d'en déduire la propriété souhaitée. Nous avons implanté cet algorithme dans notre contexte, et présenté les exemples traités, ainsi que les résultats obtenus au **chapitre 9**. Ces exemples venant de deux domaines différents : l'auto-stabilisation et la détection de terminaison.
Enfin, le **chapitre 10** présente les conclusions de ce travail ainsi que quelques perspectives.

Les résultats présentés aux chapitres 2, 5 et 8-9 ont respectivement fait l'objet des publications [DFP01], [DFP02] et [DFN01].

[5] c'est à dire par lequel toute exécution infinie va passer au moins une fois

	Chapitre 1	Introduction.
Partie I : Introduction aux probabilités et systèmes distribués	Chapitre 2	Notions de théorie des probabilités et introduction de deux modèles stochastiques : chaînes de Markov et processus de décision markoviens.
	Chapitre 3	Présentation des systèmes et algorithmes distribués, notamment sur des anneaux paramétrés.
Partie II : Convergence de systèmes paramétrés probabilistes	Chapitre 4	Méthode de preuve de convergence d'algorithmes probabilistes sur des systèmes paramétrés.
	Chapitre 5	Application de la preuve de convergence à une variante du dîner des philosophes probabilistes.
	Chapitre 6	Discussion sur l'utilisation des algorithmes sans équité.
Partie III : Convergence de systèmes paramétrés déterministes	Chapitre 7	Présentation de la méthode de preuve de convergence de Beauquier, Bérard, Fribourg et Magniette.
	Chapitre 8	Une étape vers l'automatisation de preuves de convergence grâce à la réécriture préfixe.
	Chapitre 9	Applications de notre méthode de preuve à l'auto-stabilisation et la détection de terminaison, ainsi que les résultats obtenus.
	Chapitre 10	Conclusions et perspectives.

FIG. 1.1 – Organisation de la thèse

Première partie

Introduction aux probabilités et systèmes distribués

Chapitre 2

Le formalisme probabiliste

Comme on s'intéressera dans la partie II à des systèmes probabilistes, il est nécessaire de présenter les modèles stochastiques qui seront utilisés pour décrire les systèmes probabilistes que nous allons étudier. Notre but n'est évidemment pas de donner une introduction relativement complète à la théorie des probabilités, mais de présenter les outils et méthodes dont nous aurons besoin dans la suite de cette thèse.

Ce chapitre présente donc pour commencer quelques notions de probabilités à la section 2.1, utiles tant pour la suite de ce chapitre que pour la partie II. Nous introduisons ensuite les deux modèles stochastiques que nous allons considérer. Tout d'abord les chaînes de Markov en temps discret, où tout le non-déterminisme réside dans les choix probabilistes, sont présentées à la section 2.2. Nous nous intéressons pour finir aux processus de décision markoviens, systèmes qui mêlent à la fois non-déterminisme fort et probabilités, à la section 2.3. C'est ce dernier modèle qui nous servira dans la suite à décrire les systèmes distribués probabilistes, et nous montrerons dans quel cas on peut se ramener à une chaîne de Markov.

Non déterminisme *vs* "probabilisme"

Lorsque l'on parle de non déterminisme et de probabilités, on est souvent amené à utiliser des abus de langage. En effet, formellement parlant, le choix probabiliste entre plusieurs états est une forme de non-déterminisme. Seulement, on est souvent amené à vouloir opposer le non-déterminisme "fort", c'est-à-dire le non déterminisme non probabiliste, celui où le choix entre les états se fait de manière arbitraire, et le non-déterminisme "probabiliste". C'est pourquoi dans la suite, lorsqu'il n'y aura pas d'ambiguïté, on opposera choix non-déterministe (lorsqu'il s'agit du non-déterminisme fort) et choix probabiliste.

2.1 Quelques notions de probabilités

Cette section a pour but de présenter, dans un cadre relativement formel, une base théorique suffisante pour pouvoir définir les deux modèles stochastiques que l'on va utiliser dans les chapitres suivants : chaînes de Markov et processus de décision markoviens. Elle ne prétend pas donner une introduction exhaustive de la théorie des probabilités. Les références en la matière ne manquent pas. Parmi elles on peut citer l'ouvrage [Bré99] d'où vient la terminologie utilisée ici. On peut aussi regarder l'article [Pan01] où les probabilités sont présentées pour les informaticiens, et plus précisément pour le cadre des systèmes concurrents.

Lorsque l'on veut étudier un phénomène probabiliste, on considère un ensemble particulier Ω, contenant tous les résultats possibles des observations de ce phénomène.

Définition 2.1. *L'ensemble Ω contenant tous les résultats possibles des observations d'un phénomène est appelé* ensemble des observables.

Exemple 2.2. Dans le cas d'un lancer de dé, les résultats possibles sont $\omega = 1, \omega = 2, ..., \omega = 6$ et l'ensemble des observables est $\Omega = \{1, 2, 3, 4, 5, 6\}$. ♣

Tout sous-ensemble de l'ensemble des observables peut être considéré comme la représentation d'un *événement*. Dans le cas du dé, $A = \{1, 3, 5\}$ est l'événement correspondant à un résultat impair.

La théorie des probabilités consiste à associer aux événements leur *probabilité*. Dans le cas général, on n'assigne pas de probabilité à tous les événements possibles mais à une collection d'événements, notée $\mathcal{F}$ (où $\mathcal{F} \subseteq \mathcal{P}(\Omega)$).

Sur quel ensemble définir des probabilités ?

Ce que l'on souhaite, c'est avoir une collection dans laquelle on puisse tout d'abord définir la probabilité de Ω tout entier (qui sera 1). On veut de plus que la collection soit stable par complémentaire, pour que, lorsqu'on connaît la probabilité p d'un événement, on puisse calculer la probabilité du complémentaire ($= 1 - p$). On veut également pouvoir calculer la probabilité d'une union d'événements lorsqu'on connaît la probabilité de chacun.

Formellement, on définit la probabilité sur une collection notée $\mathcal{F} \subset \mathcal{P}(\Omega)$, satisfaisant la propriété de σ-algèbre suivante :

Définition 2.3. *Soit X un ensemble, et soit $\Sigma \subset \mathcal{P}(X)$ une famille de sous-ensembles. Σ est une σ-*algèbre *ou* tribu *si elle satisfait les propriétés suivantes :*

(1) $\emptyset \in \Sigma$

(2) $(A \in \Sigma) \Rightarrow (A^c \in \Sigma)$ *(où ^{c} dénote le complémentaire)*

(3) $(\forall i \in I, A_i \in \Sigma) \Rightarrow \bigcup_{i \in I} A_i \in \Sigma$ *pour tout ensemble I dénombrable.*

En exprimant l'intersection à l'aide des opérations d'union et de complémentaire, on déduit de cette définition que l'ensemble X appartient à Σ et qu'une σ-algèbre est fermée par intersection dénombrable. Bien évidemment, étant donné un ensemble d'observables Ω, il existe plusieurs σ-algèbres associées.

Exemple 2.4. Considérons à nouveau l'exemple du lancer de dé. Imaginons que l'on ne s'intéresse qu'aux événements qui s'expriment en comparant le résultat du dé aux valeurs 1 et 2, et à l'aide d'opérateurs ensemblistes (intersection, union, complémentaire). On obtient alors la σ-algèbre suivante :

$$\mathcal{F} = \{\{1\}, \{2\}, \{1, 2\}, \{3, 4, 5, 6\}, \{1, 3, 4, 5, 6\}, \{2, 3, 4, 5, 6\}, \emptyset, \Omega\}.$$

Ces événements correspondent respectivement à $\{1\}$, $\{2\}$, $\{1\} \cup \{2\}$, $(\{1\} \cup \{2\})^c$, $\{2\}^c$, $\{1\}^c$, $\{1\} \cap \{2\}$ et $(\{1\} \cap \{2\})^c$. Dans ce cas, $\mathcal{F}$ est appelée *σ-algèbre engendrée* par les événements 1 et 2. ♣

Lorsqu'un ensemble d'observables Ω est muni d'une σ-algèbre $\mathcal{F}$, on dit que $(\Omega, \mathcal{F})$ est un *espace mesurable.*
Cette notion de σ-algèbre $\mathcal{F}$ des ensembles mesurables sera utilisée à la section 2.2.2 puis à la section 2.3.1, lorsque l'on définira une notion de probabilités sur les chemins puis sur les exécutions. Pour des ensembles d'observables plus simples (comme par exemple la probabilité qu'une règle probabiliste donne un certain résultat), on prendra $\mathcal{F} = \mathcal{P}(\Omega)$.

Relativement à la notion d'espace mesurable, on peut définir ce qu'est une fonction mesurable :

Définition 2.5. *On appelle* fonction mesurable *d'un espace mesurable* $(\Omega, \mathcal{F})$ *dans un espace mesurable* $(\Omega', \mathcal{F}')$ *toute fonction* $f : \Omega \mapsto \Omega'$ *telle que*

$$\forall E \in \mathcal{F}',\ f^{-1}(E) \in \mathcal{F}.$$

Dans la définition ci-dessus, la fonction f^{-1} désigne l'image réciproque par f, c'est-à-dire que $f^{-1}(E) = \{x \in \Omega \mid f(x) \in E\}$. Une fonction mesurable assure donc que l'image réciproque de tout ensemble mesurable de $\mathcal{F}'$ est un ensemble mesurable appartenant à $\mathcal{F}$.

La notion d'espace mesurable a été introduite dans le but d'y associer une fonction particulière, appelée mesure, qui associe une valeur à chaque ensemble mesurable. Nous allons considérer ici un cas particulier des mesures : celles de poids total 1, appelées mesures de probabilités.
La probabilité d'un événement mesure la fréquence de cet événement, la “chance” que l'on a de le constater lorsqu'on observe un phénomène probabiliste.

Définition 2.6. *On appelle* mesure de probabilité*, ou* distribution *sur un espace mesurable* $(\Omega, \mathcal{F})$ *toute application* $\mathbb{P}$ *de* $\mathcal{F}$ *dans* $\mathbb{R}^+ (= [0, +\infty[)$ *telle que :*

1. $\mathbb{P}$ *est définie pour tout élément de* $\mathcal{F}$,
2. $\mathbb{P}(\Omega) = 1$,
3. *pour tout ensemble dénombrable* $\{A_i, i \in I\}$ *d'éléments de* $\mathcal{F}$ *disjoints deux à deux, on a* $\mathbb{P}(\bigcup_{i \in I} A_i) = \sum_{i \in I} \mathbb{P}(A_i)$ *(σ-additivité).*

L'ensemble des distributions sur Ω *est noté* $\mathcal{D}(\Omega)$.
Le triplet $(\Omega, \mathcal{F}, \mathbb{P})$ *est appelé* espace de probabilités*.*

Exemple 2.7. Pour le lancer de dé, et en prenant $\mathcal{P}(\Omega)$ comme σ-algèbre, pour tout ensemble $A \subset \Omega = \{1, 2, 3, 4, 5, 6\}$, la fonction

$$\mathbb{P}(A) = \frac{|A|}{|\Omega|} = \frac{|A|}{6}$$

est une mesure de probabilité sur $(\Omega, \mathcal{P}(\Omega))$.

Toujours pour le lancer de dé mais avec la σ-algèbre $\mathcal{F}$ donnée dans l'exemple 2.4, la restriction $\mathbb{P}'$ de $\mathbb{P}$ à $\mathcal{F}$ est une mesure de probabilité. ♣

En général, lorsque l'on considère l'espace des réels, (resp. la droite $\overline{\mathbb{R}} = [-\infty, +\infty]$ des réels achevée) on lui associe une σ-algèbre particulière appelée σ-algèbre des boréliens, qui est la plus petite σ-algèbre engendrée[1] par les intervalles de la forme $]a, b]$ avec $-\infty \leq a < b < \infty$ (resp. avec $-\infty \leq a < b \leq \infty$). L'espace mesurable associé, dit de Borel, est noté $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$ (resp. $(\overline{\mathbb{R}}, \mathcal{B}(\overline{\mathbb{R}}))$), ou plus simplement $(\mathbb{R}, \mathcal{B})$ (resp. $(\overline{\mathbb{R}}, \mathcal{B})$).

[1] pour une preuve de l'existence d'une σ-algèbre engendrée par un ensemble, voir par exemple [KSK66]

Définition 2.8. *Une* variable aléatoire *sur un espace de probabilités* $(\Omega, \mathcal{F}, \mathbb{P})$ *est une fonction mesurable* X *de* $(\Omega, \mathcal{F})$ *dans* $(\overline{\mathbb{R}}, \mathcal{B})$.

Une variable aléatoire sert à associer aux résultats observables $\omega \in \Omega$ une certaine valeur, fonction de ω, qui va nous intéresser.

Exemple 2.9. Supposons que l'on s'intéresse toujours à un lancer de dés, mais cette fois ci dans le cadre d'un jeu, le but étant de faire le plus grand "score" avec les dés. On va alors associer à chaque résultat un score (par exemple la somme des valeurs des dés), et la fonction

$$X: \begin{array}{ccc} \Omega & \mapsto & \mathbb{N} \\ (a,b,c) & \rightarrow & a+b+c \end{array}$$

est une variable aléatoire. ♣

Il reste encore à définir une notion qui sera cruciale pour la définition des chaînes de Markov, celle de *probabilité conditionnelle.*

Définition 2.10. *Soient* $(\Omega, \mathcal{F}, \mathbb{P})$ *un ensemble de probabilités,* A *et* B *deux événements. On appelle* probabilité de A sachant B *et on note* $\mathbb{P}(A|B)$ *la probabilité définie (uniquement quand* $\mathbb{P}(B) > 0$*) par :*

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}$$

Deux événements sont dits *indépendants* si la probabilité que A soit vrai ou non n'est pas influencée par le fait que B soit vrai. On a alors $\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B)$ et $\mathbb{P}(A|B) = \mathbb{P}(A)$.

Exemple 2.11. Toujours pour l'expérience de lancer du dé, considérons les trois événements suivants :
A : ω est pair
B : ω est multiple de 3
C : ω est inférieur ou égal à 3
On a, pour la mesure de probabilité $\mathbb{P}$ définie à l'exemple 2.7, $\mathbb{P}(A) = 3/6$, $\mathbb{P}(B) = 2/6$ et $\mathbb{P}(C) = 3/6$. Si on calcule les probabilités conditionnelles associées on a :

$$\mathbb{P}(A|B) = \frac{1/6}{2/6} = \mathbb{P}(A), \quad \mathbb{P}(A|C) = \frac{1/6}{3/6} \neq \mathbb{P}(A), \text{ et } \mathbb{P}(B|C) = \frac{1/6}{3/6} = \mathbb{P}(B).$$

On constate donc que les événements A et B sont indépendants, de même que les événements B et C, mais pas les événements A et C.
De plus, B est indépendant des événements A et C pris séparément, mais clairement pas de l'événement $A \cap C$, car $\mathbb{P}(B|A \cap C) = 0$. ♣

A l'aide de toutes les notions présentées ci-dessus, nous sommes à présent en mesure d'introduire les deux modèles probabilistes que nous allons utiliser dans la suite de cette thèse : chaînes de Markov et processus de décision markoviens

2.2 Chaînes de Markov en temps discret

Les notions relatives aux chaînes de Markov présentées dans cette partie sont tirées des ouvrages [Bré99] et [Nor98].

2.2.1 Premières définitions

Soit S un ensemble dénombrable. Dans la suite on va appeler les éléments $s \in S$ des *états* et S l'*ensemble d'états*. En utilisant les notions de la section précédente, on peut définir ce qu'est une chaîne de Markov.

Définition 2.12. *Soit $(X_n)_{n\geq 0}$ une suite de variables aléatoires à valeurs dans S. $(X_n)_{n\geq 0}$ est une* chaîne de Markov *si $\mathbb{P}(X_n = j|X_0 = i_0, ..., X_{n-1} = i) = \mathbb{P}(X_n = j|X_{n-1} = i)$ pour tout $n > 0$ et tout $n+1$-uplet d'états de S $i_0, ..., i_{n-2}, i, j$ pour lequel les deux membres ci-dessus sont définis.*

Une chaîne de Markov est donc une suite de variables aléatoires telle que la distribution d'une variable ne dépend que de celle de la variable précédente, et pas de tout l'historique.

Une chaîne de Markov est dite *homogène* si $\mathbb{P}(X_n = j|X_{n-1} = i)$ ne dépend que de i et j, et pas de n. Dans ce cas on peut définir la *matrice de transition* $T = (t_{ij})_{i,j\in S}$ associée à la chaîne de Markov par $t_{ij} = \mathbb{P}(X_n = j|X_{n-1} = i)$. Cette matrice n'est pas quelconque : au contraire, elle vérifie que tout ses coefficients sont positifs, et que, pour chaque ligne, la somme des coefficients vaut 1 : $\forall i \in S, \sum\limits_{j\in S} t_{ij} = 1$. Une telle matrice est appelée *matrice stochastique*.

Dans toute la suite on va toujours considérer, sans le préciser, des chaînes de Markov homogènes.

Pour se rapprocher du cadre informatique, on va considérer non plus une véritable chaîne de Markov, mais un *système markovien* à savoir un système dont le comportement suit celui d'une chaîne de Markov. À toute matrice de transition d'une chaîne de Markov correspond un graphe représentant un système markovien. Dans ce graphe, il existe une transition $s_i \xrightarrow{p} s_j$ de l'état s_i à l'état s_j étiquetée par $p > 0$ si et seulement si $t_{ij} = p$ dans la matrice de transition. La figure 2.1 représente une matrice de transition et le graphe de transition du système markovien associé.

$$T = \begin{pmatrix} 0 & 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/4 & 1/8 & 1/8 & 1/2 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1/2 & 0 & 0 & 1/2 \end{pmatrix}$$

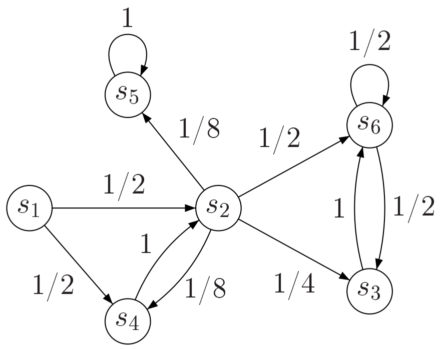

FIG. 2.1 – Un exemple de graphe associé à une chaîne de Markov

Un système markovien va alors être envisagé comme un système de transitions (*i.e.* un graphe orienté) probabiliste dans lequel, à chaque étape, on se déplace en tirant au sort le prochain état, et ce suivant les probabilités associées aux transitions.

Formellement, un *système de transitions probabiliste* est un graphe étiqueté tel que les étiquettes sont à valeurs dans $]0,1]$, et tel que pour tout état s du graphe, la somme des étiquettes des transitions partant de s vaut 1.

La représentation sous forme de graphe montre bien que, partant d'un certain état s_i, la probabilité d'aller en une étape dans un autre état s_j ne peut dépendre que de l'état s_i. Quand on arrive dans un état, on perd toute information sur comment on y est parvenu.

2.2.2 Probabilités sur les chemins

En se basant sur cette nouvelle caractérisation sous forme de systèmes de transitions probabilistes, on va pouvoir définir ce qu'est un chemin.

Définition 2.13. *Étant donné un système de transitions probabiliste G, un* chemin *(resp. un* chemin fini*) dans G est une suite infinie (resp. finie) d'états $c = s_0, s_1, ..., s_n, ...$ (resp. $s_0, s_1, ..., s_k$) telle que, pour tout i (resp. tout $i \in \{0, .., k-1\}$), (s_i, s_{i+1}) est une arête de G.*

Lorsqu'un système est markovien, la probabilité d'aller en une transition de l'état s à l'état s' est définie sans ambiguïté. On peut donc définir des probabilités sur les chemins.

Pour un chemin fini, on va prendre pour probabilité le produit des probabilités des transitions qui le composent, mais lorsque l'on a un chemin (infini), la définition de l'espace de probabilités nécessite plus de précautions.

Étant donnée une relation de transition probabiliste, on définit une relation d'accessibilité en un pas : $Acc \subset S \times S$ telle qu'on a $Acc(s,t)$ si et seulement si il existe une transition de probabilité p (non nulle) entre s et t.

Pour tout état $s \in S$ on définit alors

$$\Omega_s = \{s_0 s_1 s_2 ... | s = s_0 \text{ et } \forall n \in \mathbb{N}, Acc(s_n, s_{n+1})\}$$

qui est l'ensemble de tous les chemins (infinis) issus de s.

Définition 2.14. *Étant donnée une suite finie $c = s, s_1, ..., s_k$ d'états de S on définit le* cylindre *de base c et on note $\mathcal{C}_c$ l'ensemble des chemins de Ω_s qui commencent par le chemin fini $s, s_1, ..., s_k$, autrement dit l'ensemble des chemins $s'_0, s'_1, s'_2, ..., s'_n, ...$ de Ω_s tels que $s'_1 = s_1, ..., s'_k = s_k$.*

Ici on n'a pas besoin de préciser $s'_0 = s$, car comme on prend des chemins de Ω_s, le premier état est fixé et vaut s. Un exemple de cylindre est donné à la figure 2.2.

Dans la littérature, on trouve également la notion de *cône* (cf. [BDLGJ02, SL95]) pour définir ces ensembles particulier de chemins. Nous lui préférons celle de cylindre de [KSK66], voir aussi [BdA95].

Soit G un système de transitions associé à une chaîne de Markov de matrice de transition T. On définit la distribution $\mathbb{P}$ sur l'ensemble des cylindres de la façon suivante :

Soit $\mathcal{C}_c$ un cylindre de base $c = s_0, s_1, ..., s_k$. La probabilité $\mathbb{P}(\mathcal{C}_c)$ est définie par :

$$\mathbb{P}(\mathcal{C}_{s_0,...,s_k}) = \prod_{0 \leq i \leq k-1} t_{s_i s_{i+1}}$$

.

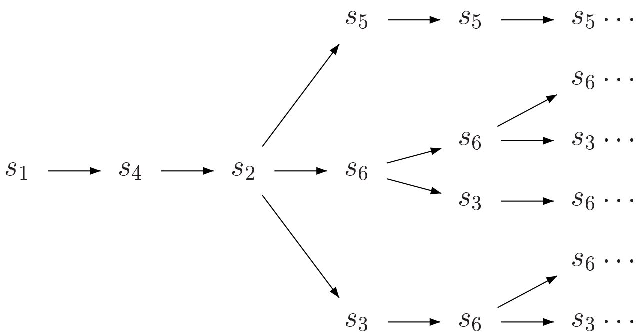

FIG. 2.2 – Ensemble des chemins d'un cylindre de base s_1, s_4, s_2

Exemple 2.15. Reprenons le système donné à la figure 2.1. Le cylindre de base s_1, s_4, s_2 a comme probabilité $1/2$ $(= 1/2 \times 1)$ et désigne l'ensemble de chemins représenté à la figure 2.2. ♣

On considère donc pour tout état s l'espace de probabilités $(\Omega_s, \mathcal{F}_s, \mathbb{P}_s)$ où $\mathcal{F}_s$ est la σ-algèbre engendrée par les cylindres dont la base est un chemin fini partant de s, et $\mathbb{P}_s$ est l'extension à $\mathcal{F}_s$ de la mesure de probabilités définie ci-dessus. Pour des définitions plus précises, entre autres de l'existence d'une σ-algèbre engendrée, voir par exemple [KSK66].

On a ainsi caractérisé les ensembles de chemins *mesurables*, c'est-à-dire ceux sur lesquels est définie la probabilité. Cette caractérisation est de plus pertinente car elle contient tous les ensembles de chemins définis par des formules de logique probabiliste que l'on voudra exprimer par la suite, comme par exemple l'ensemble des chemins qui passent par un ensemble donné, qui servira pour prouver la convergence.

2.2.3 États récurrents, états transitoires

A l'aide de notre définition de la mesure de probabilité sur les chemins, on peut définir ce qu'est un état récurrent.

Définition 2.16. *Un état s d'un système markovien est* récurrent *si la probabilité de revenir infiniment souvent dans cet état vaut* 1, *c'est à dire :*

$$\mathbb{P}_s(\{s_0, s_1, ..., s_n, ... \in \Omega_s | s_i = s \textit{ pour une infinité d'indices } i\}) = 1.$$

Si ce n'est pas le cas, l'état est dit transitoire. *$\mathcal{R}ec$ est une notation pour l'ensemble des états récurrents.*

L'ensemble $\{s_0, s_1, ..., s_n, ... \in \Omega_s | s_i = s$ pour une infinité d'indices $i\}$ peut être décrit à l'aide d'unions et d'intersections dénombrables de cylindres, et la probabilité ci-dessus est bien définie.

Si on se restreint comme on va le faire dans les chapitres suivants au cas où l'ensemble d'états du système markovien est fini, alors on peut caractériser les états transitoires et récurrents de manière différente, en utilisant le graphe associé.

On va adopter la notation $s \to s'$ pour dire qu'il existe p strictement positif tel que $s \xrightarrow{p} s'$. On définit la fermeture transitive $\to^*$ de la relation $\to$ par

$$s \to^* s' \Leftrightarrow \exists s_0, ..., s_n \in S : \ s_0 = s, s_n = s' \text{ et } \forall i \in \{0, .., n-1\}, \ s_i \to s_{i+1}.$$

On a alors :

Proposition 2.17. *Pour un système markovien fini, un état s est récurrent ssi*

$$(i) \quad \forall s' \in S(s \to^* s' \Rightarrow s' \to^* s).$$

Il est transitoire ssi

$$(ii) \quad \exists s' \in S : \ (s \to^* s' \wedge \neg(s' \to^* s)).$$

La justification de cette classification des états d'un système markovien en états transitoires d'une part et états récurrents d'autre part sera donnée à la section 2.2.4 quand on aura défini la convergence probabiliste.

On peut constater que les deux propriétés (i) et (ii) ci-dessus forment une partition de l'ensemble des états. Cependant, dans le cas où l'ensemble d'états est infini dénombrable, la proposition ci-dessus ne peut pas s'appliquer, comme on peut le constater sur l'exemple figure 2.3. Tout état y satisfait (i) mais un calcul montre que tout état est transitoire : avec probabilité 1 on va diverger vers l'infini à droite.

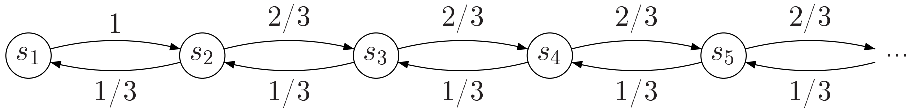

FIG. 2.3 – Une chaîne de Markov infinie sans états récurrents

Une autre caractéristique des états peut être importante dans le cadre de systèmes markoviens, c'est la notion d'états absorbants (que l'on utilisera à la section 4.4) :

Définition 2.18. *Un état α d'une chaîne de Markov est* absorbant *si la probabilité de sortir de cet état est nulle, à savoir $\mathbb{P}(X_{n+1} = \alpha | X_n = \alpha) = 1$. L'ensemble des éléments absorbants est noté $\mathcal{A}bs$.*

En termes de système markovien, une fois qu'on arrive dans un état absorbant, on ne peut plus en sortir. Une chaîne de Markov est dite *absorbante* si tous les éléments récurrents sont absorbants et réciproquement ($\mathcal{R}ec = \mathcal{A}bs$).

On peut remarquer que, par définition, tout état absorbant est récurrent.

Si on reprend la caractérisation des chaînes de Markov par des graphes, on peut caractériser les états récurrents en termes de composantes fortement connexes. En effet, deux états s et s' d'un graphe appartiennent à la même *composante fortement connexe* si on a $s \to^* s'$ et $s' \to^* s$. L'ensemble des composantes fortement connexes d'un graphe forme une partition des sommets.

En reprenant la caractérisation des états récurrents donnée à la proposition 2.17, on constate que la propriété (i) est équivalente à :

Proposition 2.19. *Un état est récurrent si et seulement si il appartient à une composante fortement connexe fermée (c'est à dire dont on ne peut pas sortir).*

Démonstration. Tout d'abord tout état appartient à une composante fortement connexe (que nous appellerons dans cette preuve CFC), et une seule. Il suffit donc de montrer que si elle est fermée l'état est récurrent, et si elle ne l'est pas, l'état est transitoire. Supposons que s appartienne à une CFC fermée. On sait alors pour tout état s', si $s \to^* s'$ alors l'état s'

appartient à la CFC de s, et donc $s' \to^* s$. Comme c'est vrai pour tout s', s est récurrent. Supposons maintenant que la CFC de s ne soit pas fermée. Il existe donc s' appartenant à la CFC de s et s'' hors de la CFC de s tels que $s' \to s''$. Or par définition des CFC on a $s \to^* s'$ d'où $s \to^* s''$. Comme s'' n'est pas dans la CFC de s et qu'on vient de montrer que $s \to^* s''$, on ne peut pas avoir $s'' \to^* s$, ce qui prouve que s est transitoire. □

On peut également remarquer la particularité des états absorbants quant aux composantes fortement connexes : tout état absorbant est seul dans sa composante fortement connexe.

2.2.4 Temps moyen de convergence

Définition 2.20. *Étant donné un système markovien sur un espace d'états S, et E un sous ensemble de S. On dit qu'il y a* convergence probabiliste *du système vers l'ensemble E si pour tout état s, la probabilité de l'ensemble des chemins issus de s qui passent par E vaut 1. Formellement cela s'écrit :*

$$\forall s \in S,\ \mathbb{P}_s(c = s_0...s_n... \in \Omega_s | \exists i :\ s_i \in E) = 1.$$

On utilise le terme de convergence probabiliste, car ici on veut montrer que tous les chemins sauf un ensemble de probabilité nulle passent par l'ensemble E.

On peut remarquer, comme on l'avait annoncé avant, que l'ensemble des chemins caractérisé ci-dessus est bien dans $\mathcal{F}_s$ car c'est une union dénombrable d'unions finies de cylindres :

$$\bigcup_{i\in\mathbb{N}} (\bigcup_{t_1\in S} ... \bigcup_{t_{i-1}\in S} \bigcup_{t\in E} \{c = s_0...s_n... \in \Omega_s | s_1 = t_1, ..., s_{i-1} = t_{i-1}, s_i = t\})$$

On remarque que la quantification ne porte pas sur s_0. Ceci est dû au fait que, comme on considère des chemins de Ω_s, s_0 est fixé et vaut s.

L'intérêt majeur de classifier les états en états récurrents d'une part, et états transitoires d'autre part réside dans le résultat suivant :

Théorème 2.21. (Markov) *Soit $\mathcal{S}$ un système se comportant comme une chaîne de Markov, on a convergence probabiliste de $\mathcal{S}$ vers l'ensemble $\mathcal{R}ec$ des états récurrents.*

Pour un état $s \in S$ donné, on a vu qu'on a un espace probabiliste $(\Omega_s, \mathcal{F}_s, \mathbb{P}_s)$. On a donc en particulier un ensemble mesurable $(\Omega_s, \mathcal{F}_s)$.
Étant donné un ensemble de configurations E, considérons la fonction X suivante, qui compte le nombre d'étapes le long d'un chemin nécessaires pour atteindre l'ensemble E :

$$\begin{array}{cccc} X : & \Omega_s & \mapsto & \overline{\mathbb{R}} \\ & c = s_0 s_1 ... s_n ... & \to & i \text{ tel que } s_i \in E \wedge \forall j < i, s_j \notin E \end{array}$$

Cette fonction est une variable aléatoire. En effet, comme elle prend ses valeurs dans $\{\mathbb{N} \cup \infty\}$, pour tout intervalle $]a, b]$ de $\overline{\mathbb{R}}$ qui permet d'engendrer la σ-algèbre des boréliens (voir section 2.1) $X^{-1}(]a, b])$ est une union au plus dénombrable de $X^{-1}(c)$ où $c \in \mathbb{N} \cup \infty$. Or, $X^{-1}(\{c\})$ est une union finie de cylindres, dont la base est de longueur c, dont les $c - 1$ premiers états ne sont pas dans E mais le $c^{\text{ème}}$ est dans E.

En se basant sur les considérations ci-dessus, le *temps moyen de convergence* d'un système markovien se calcule en faisant une "moyenne" sur l'ensemble des chemins de la longueur de ces chemins (la valeur de $X(\omega)$), cette moyenne étant pondérée par la probabilité de ces chemins. Cette moyenne, appelée aussi *espérance* de la variable aléatoire X, peut se calculer de la façon suivante :

$$E(X) = \sum_{k \in \mathbb{N} \cup \infty} k \times \mathbb{P}(X^{-1}(\{k\}))$$

Bien sûr, pour que cette somme soit finie il faut que l'ensemble des chemins de $X^{-1}(\{\infty\})$ soit de probabilité nulle, la condition ci-dessus ($\mathbb{P}(X^{-1}(\{\infty\})) = 0$) étant nécessaire mais pas suffisante.

Dans la section 4.4, nous allons nous intéresser au temps moyen de convergence d'un système markovien, et pour cela nous allons appliquer un résultat permettant de calculer ce temps moyen directement à partir de la matrice de transition, et sans calculer explicitement les ensembles $X^{-1}(\{c\})$ (voir la section 4.4 pour plus de détails).

2.3 Processus de décision markoviens

Pour modéliser certains systèmes probabilistes, il est parfois également nécessaire de considérer des comportements non-déterministes, et ce pour plusieurs raisons. Tout d'abord ce comportement peut venir du système lui-même, par exemple dans le cas de systèmes distribués asynchrones, où chaque processus évolue à son rythme et il n'est pas possible de prévoir *a priori* dans quel ordre les processus vont agir. Le non-déterminisme peut être lié au fait que, par exemple pour simplifier un système, on ne spécifie pas tous les comportements possibles, ou toutes les restrictions sur ces comportements. On peut aussi utiliser le non-déterminisme lorsque l'on ne connaît pas précisément toute la spécification du système, ou qu'on ne veut pas la fixer pour le moment. Le système peut également être en interaction avec son environnement, que l'on ne maîtrise pas et qui, du point de vue du système a un comportement non-déterministe.

Il est donc nécessaire pour étudier ce genre de systèmes d'introduire un formalisme combinant probabilités et non-déterminisme. Les *processus de décision markoviens* (dénotés dans la suite PDM), appelés aussi dans la littérature *Probabilistic Nondeterministic Systems* (cf. [BdA95]) permettent de mêler choix non-déterministes et choix probabilistes de la manière suivante :

Définition 2.22. *On appelle* processus de décision markovien *fini un quadruplet* $\mathcal{M} = (S, Acts, A, p)$ *où :*
- S *est un ensemble fini d'états,*
- $Acts$ *est un ensemble fini d'actions,*
- $A : S \mapsto 2^{Acts}$ *est une fonction qui associe à chaque état* $s \in S$ *l'ensemble des actions qui sont possibles dans cet état,*
- $p : S \times Acts \mapsto \mathcal{D}(S)$ *qui associe à tout couple* (s, a) *une mesure de probabilités sur* S. *pour tout état* t, $p(s, a)(t)$ *est la probabilité de prendre une transition de* s *à* t *lorsque l'action* a *est choisie.*

Cette définition, tirée de [dA99] est une version simplifiée de celle donnée dans [Put94]. En effet ici on ne s'intéresse pas à l'intervalle de temps entre deux transitions, mais c'est le nombre de transitions effectuées qui va constituer notre mesure de "temps". Comme pour les chaînes de Markov que l'on va considérer, dans la suite et même si l'on ne le précise pas, l'ensemble d'états de tout PDM sera fini.

Cette idée de choix non déterministe d'une action suivi d'un choix probabiliste de l'état suivant correspond à la notion de *systèmes probabilistes réactifs* (par opposition aux modèles génératifs ou stratifiés) présentés dans [vGSS95].

D'autres modèles mêlant non-déterminisme et probabilités ont été étudiés, comme les automates probabilistes [Rab63, SL95]. La thèse d'habilitation [Bai98] présente les systèmes concurrents probabilistes, et donne de nombreuses références sur d'autres systèmes probabilistes et non-déterministes.

Le non-déterminisme réside ici uniquement dans le choix d'une action. Étant donnés un état et une action, l'état suivant est choisi suivant une mesure de probabilités.

Définition 2.23. *Une* exécution *du PDM $\mathcal{M}$ (resp. une* exécution finie*) est une suite infinie (resp. finie), d'états et d'actions alternés $\rho : s_0, a_0, s_1, a_1, \ldots$ telle que, pour tout i (resp. tout $i < |\rho| - 1$), on a $p(s_i, a_i)(s_{i+1}) > 0$. Lorsque l'exécution est finie, le dernier état s_n de ρ est appelé état d'arrivée.*

Entre deux états consécutifs de l'exécution, il y a eu tout d'abord un choix non-déterministe entre les actions puis un choix probabiliste entre les états successeurs potentiels. Dans la suite on notera indifféremment une exécution soit comme une suite alternée d'états et d'actions $s_0, a_0, s_1, a_1, \ldots$, soit, pour coller à la représentation par un graphe, comme une suite de transitions $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$.

Avec de tels systèmes, à cause du non-déterminisme sur le choix des actions, on ne peut plus définir de manière exacte par exemple la probabilité $\mathbb{P}_s(\Delta)$ d'emprunter en partant de s une exécution d'un sous-ensemble donné Δ d'exécutions. La définition de notre nouvel espace de probabilités nécessite des définitions plus précises qui sont présentées dans la section suivante.

2.3.1 Espaces de probabilités pour un processus de décision markovien

Lorsque l'on a un PDM il n'est pas possible, contrairement au cas des chaînes de Markov, de définir directement des probabilités sur les chemins. Comme le choix entre les actions se fait de manière fortement non-déterministe, on a un ensemble d'espaces de probabilités à considérer (un pour chaque arbre de choix non-déterministes)

On a donc besoin de restreindre l'ensemble des chemins considérés, c'est à dire de déterminer d'une manière ou d'une autre les actions qui vont être choisies à chaque étape, afin de pouvoir définir un espace de probabilités, ce qui est fait au moyen des ordonnancements. D'autre part on ne peut à présent plus définir de probabilité directement sur les chemins : la connaissance d'une suite d'états n'est plus suffisante pour savoir avec quelle probabilité ce chemin est emprunté, il faut également savoir quelles actions ont été choisies. On va alors définir l'espace de probabilités sur l'ensemble des exécutions, et non plus des chemins.

Ordonnancement

À partir d'un processus de décision markovien, il existe une façon de se ramener à un contexte où l'on peut définir des probabilités sur les chemins : il suffit de fixer les choix non-déterministes. Pour définir les choses proprement, on a besoin de la notion d'*ordonnancement* (appelé *policy* dans [dA99]). D'autres notions assez proches sont utilisées dans la littérature, comme celle d'*adversary* (pour les automates probabilistes de Segala et Lynch [SL95, Seg95]) ou *scheduler* dans le cas des systèmes distribués (cf. [LR81, PZ86]), ou encore *strategy* ([BDLGJ02]).

Définition 2.24. *Un* ordonnancement *est une application qui à toute exécution finie* $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n$ *associe une mesure de probabilité sur l'ensemble des actions possibles à partir de* s_n.

Un ordonnancement $\mathcal{O}$ associe donc à une exécution finie ρ et une action a la probabilité $\mathcal{O}(\rho)(a)$ de choisir a après avoir parcouru l'exécution ρ. Bien évidemment l'ordonnancement ne peut donner une probabilité positive qu'aux actions possibles dans l'état d'arrivée s_n de ρ.

Définition 2.25. *On dit qu'un ordonnancement est* déterministe *si* $\mathcal{O}(\rho)(a) = 0$ *ou* 1 *pour toute action* a *et toute exécution finie* ρ.
Un ordonnancement est sans mémoire *(ou* markovien*) si, dès lors que deux exécutions finies* ρ *et* ρ' *ont le même état d'arrivée* s*, alors on a* $\mathcal{O}(\rho)(a) = \mathcal{O}(\rho')(a) = \mathcal{O}(s)(a)$ *(où* s *est une exécution à un seul état).*

Ces ordonnancements particuliers jouent un rôle important car Bianco et de Alfaro ont montré que pour vérifier certains types de propriétés, il suffit de considérer des ordonnancements déterministes [BdA95], comme on le verra à la section 3.8.1.

Les ordonnancements seront réutilisés dans la partie II lorsque l'on considérera des systèmes distribués probabilistes. Dans ce contexte de systèmes distribués, le choix d'une action à chaque étape correspondra au choix des processus qui vont effectuer un changement de leur état.

Mesure de probabilités sur les exécutions

En fixant un ordonnancement, on vient de voir qu'il n'y a plus de non-déterminisme autre que dans les probabilités. On obtient alors un graphe probabiliste (éventuellement infini dans le cas où l'ordonnancement n'est pas sans mémoire), et on peut alors définir une mesure de probabilités en se basant sur des cylindres, comme on l'avait fait pour les systèmes markoviens. Cette fois-ci, la probabilité d'une transition dépend à la fois de l'état courant et de l'action choisie, et donc les probabilités sont définies sur les exécutions.

Pour définir un espace de probabilités, il faut donc fixer et l'état de départ (s), et l'ordonnancement ($\mathcal{O}$). On a alors un ensemble de chemins $\Omega_{s,\mathcal{O}}$ sur lequel on définit un espace de probabilités $(\Omega_{s,\mathcal{O}}, \mathcal{F}_{s,\mathcal{O}}, \mathbb{P}_{s,\mathcal{O}})$ tel que la σ-algèbre $\mathcal{F}_{s,\mathcal{O}}$ est celle engendrée par des cylindres de base $s, a_0, \dots a_{n-1}, s_n$ où à chaque étape, le choix de l'action et de l'état suivant est fait suivant $\mathcal{O}$.

Dans ce cas précis des processus de décision markoviens, la probabilité d'un cylindre doit prendre en compte non seulement les probabilités associées aux actions (notées $t_{s_i s_{i+1}}$ dans le cas des chaînes de Markov) mais également la probabilité que l'ordonnancement choisisse d'effectuer cette action. La probabilité d'une transition $s_n \xrightarrow{a_n} s_{n+1}$ est donc le produit de la probabilité (étant donné l'historique) que l'ordonnancement choisisse l'action a_n par la probabilité que l'action a_n donne le résultat souhaité (s_{n+1}).

On peut par exemple, étant donnés un processus de décision markovien $\mathcal{M}$, un ordonnancement $\mathcal{O}$ de $\mathcal{M}$ et un état s de $\mathcal{M}$, définir la probabilité de l'ensemble des exécutions partant de s qui convergent vers un certain ensemble de configurations E. Cette probabilité, définie formellement par $\mathbb{P}(\{c = s_0, a_0, s_1, a_1, \dots \in \Omega_{s,\mathcal{O}} \mid \exists i \in \mathbb{N} : \ s_i \in E\})$, sera notée $\mathbb{P}(s \underset{\mathcal{S}}{\overset{\mathcal{O}}{\longrightarrow}}{}^{*} E)$ dans la partie II.

Étant donnés un processus de décision markovien $\mathcal{M}$, un ordonnancement $\mathcal{O}$ et un état de départ s, on va pouvoir représenter l'espace des exécutions de $\mathcal{M}$ partant de s pour l'ordonnancement $\mathcal{O}$ sous forme d'un arbre, un peu comme on l'a fait pour le cylindre de la figure 2.2).

Formellement, on construit un arbre (infini) étiqueté $T(\mathcal{O}, s)$ appelé *arbre des exécutions* tel que :

1. la racine est étiquetée par le nom s et la probabilité 1
2. tout chemin partant de la racine correspond à une exécution possible du système $\mathcal{M}$ suivant l'ordonnancement $\mathcal{O}$, et
3. tout sommet nommé v étiqueté par s' est également étiqueté par la probabilité $\varpi(v)$ que le système suivant l'ordonnancement $\mathcal{O}$ aille de s à s' en suivant le chemin de l'arbre (c'est-à-dire le produit des probabilités des transitions qui composent ce chemin).

Cette notion d'arbre des exécutions va être utilisée à la section 4.4 lorsque l'on voudra décrire les exécutions qui convergent vers un ensemble voulu, afin de calculer le temps moyen de convergence.

Définition 2.26. *On appelle* probabilité maximale*, et* probabilité minimale*, d'un ensemble d'exécutions* $\Delta \in \mathcal{F}_s$*, les fonctions définies par*

$$\mathbb{P}_s^+ = \sup_{\mathcal{O}} \ \mathbb{P}_{s,\mathcal{O}}(\Delta), \ et \ \mathbb{P}_s^- = \inf_{\mathcal{O}} \ \mathbb{P}_{s,\mathcal{O}}(\Delta)$$

Ces probabilités minimales et maximales seront utiles lorsque l'on voudra montrer des propriétés du genre : "pour tout ordonnancement, la probabilité de passer par l'état s est inférieure à $1/2$." (voir section 3.8.1).

2.3.2 D'un processus de décision markovien à une chaîne de Markov

Il est à noter que, même lorsqu'on fixe un ordonnancement pour un processus de décision markovien fini, on n'a pas toujours une chaîne de Markov finie. En effet, il est possible qu'un ordonnancement ne fasse pas toujours le même choix lorsque l'exécution repasse plusieurs fois par le même état. On n'a donc pas un processus sans mémoire et a fortiori pas une chaîne de Markov finie. Dans certaines études, on considère qu'on a toujours une chaîne de Markov, mais en prenant comme espace d'état l'ensemble (infini) des exécutions finies. Ici, on ne veut considérer que des chaînes de Markov finies, et un ordonnancement avec mémoire ne nous permet pas de nous y ramener.

Par contre, si pour un PDM on se fixe un ordonnancement sans mémoire, on a alors une chaîne de Markov.
En effet, étant donné un état du PDM, le choix de l'état suivant se fait par deux choix probabilistes successifs : tout d'abord le choix de l'action et ensuite, en fonction de l'état courant et de l'action sélectionnée, le choix de l'état suivant.
La matrice de transition de la chaîne de Markov ainsi obtenue se calcule donc en faisant le produit de deux matrices stochastiques :

- D'une part la matrice stochastique permettant de choisir (de manière probabiliste ou déterministe) une action en fonction de l'état actuel. Elle possède $|S|$ lignes et $|S| \times |A|$ colonnes (une pour chaque couple (état,action)). Le coefficient de la ligne s_i et de la colonne (s_j, a_k) vaut
$$\begin{cases} \mathcal{O}(s_j)(a_k) & \text{si } s_i = s_j \\ 0 & \text{sinon} \end{cases}$$
- D'autre par la matrice stochastique à $|S| \times |A|$ lignes et $|S|$ colonnes, dont le coefficient sur la ligne (s_i, a_k) et colonne s_j vaut $p(s_i, a_k)(s_j)$ c'est-à-dire la valeur de la distribution $p(s_i, a_k)$ (caractérisant le PDM) dans l'état s_j.

Dans le cas particulier où l'ordonnancement est déterministe et sans mémoire, la première matrice ne contient que des 0 et un 1 par ligne. Le produit se fait alors simplement et on a, pour la matrice de transition T de la chaîne de Markov obtenue, $T_{ij} = p(s_i, a_{s_i})(s_j)$ où a_{s_i} est l'action choisie de manière déterministe à partir de s_i.

Chapitre 3

Systèmes distribués

Après avoir introduit toutes les notions utiles pour comprendre les aspects probabilistes de la thèse, nous allons nous intéresser dans ce chapitre au type de systèmes que nous allons considérer : les systèmes distribués.

Ce chapitre s'organise comme suit. La section 3.1 présente tout d'abord brièvement le modèle des systèmes distribués, les différents avantages que peuvent apporter ces systèmes ainsi que quelques problèmes classiques étudiés en algorithmique distribuée. Nous nous intéressons plus particulièrement à la section 3.1.5 à la tolérance aux fautes, et comment la mettre en œuvre dans les systèmes distribués.
La section 3.2 décrit plus précisément le modèle et la façon dont les différentes entités communiquent entre elles.
Nous précisons ensuite le type de topologie des systèmes que nous allons étudier à la section 3.3. Cette topologie particulière en anneau va nous permettre de représenter les configurations par des mots et d'étudier notre système distribué comme un système de réécriture, comme expliqué à la section 3.4.
Pour étudier précisément un système distribué, on a besoin de savoir dans quel ordre les processus vont agir. C'est pourquoi nous introduisons la notion de démon, d'ordonnancement ainsi que les différentes caractéristiques qu'ils peuvent avoir à la section 3.5.
Nous donnons à la section 3.6 un aperçu de la vérification de tels systèmes, au travers des différents types de propriétés que l'on peut chercher à vérifier. Nous nous attachons plus particulièrement aux propriétés de convergence et d'auto-stabilisation, qui sont celles que nous allons considérer dans le reste de la thèse.
La section 3.7 donne quelques pistes concernant la vérification de systèmes de taille paramétrée, à savoir ceux pour lesquels on ne fixe pas à l'avance le nombre de processus. Ce sont ces systèmes, et plus particulièrement ceux à topologie en anneau que nous allons étudier tout au long de cette thèse.
Enfin, à la section 3.8 nous présentons le modèle des systèmes distribués probabilistes, indiquant ce qui change par rapport au cas déterministe, ainsi qu'un résultat permettant de restreindre l'ensemble des ordonnancements considérés.

3.1 Introduction

Un *système distribué*, également appelé *système réparti*, se définit de la manière suivante. Il se compose d'un ensemble d'entités autonomes (ordinateurs, processeurs, automates, processus...) reliés au moyen de canaux de communication ou de variables partagées. Ils peuvent donc communiquer entre eux et évoluer en fonction de leur propre état et des informations échangées. On utilise souvent le formalisme des graphes pour décrire la topologie de ces systèmes. Ainsi, les entités sont appelées les *nœuds* du système et on utilise des arêtes pour représenter les communications entre les nœuds. Un exemple est donné à la figure 3.1.

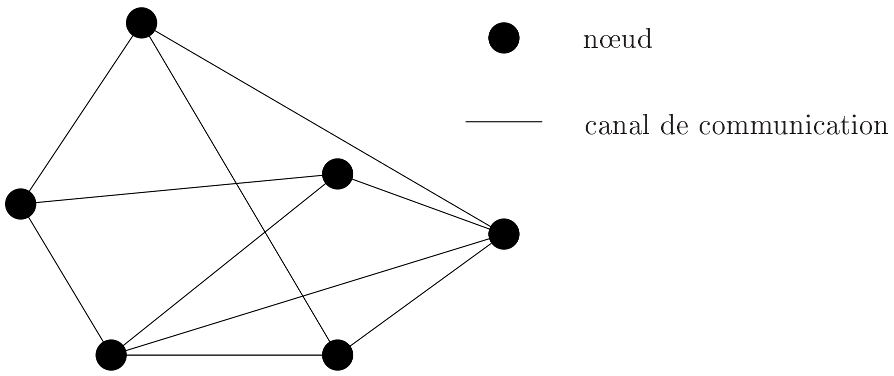

FIG. 3.1 – Exemple de réseau associé à un système distribué.

De nos jours on ne met plus en doute l'utilité des systèmes distribués et des algorithmes qui leur sont associés. Ceux-ci sont présents dans des applications aussi diverses que les réseaux (qu'ils soient locaux, ou longue distance comme Internet), les ordinateurs multiprocesseurs, ou certains systèmes critiques dans lesquels on utilise plusieurs modules effectuant la même tâche pour augmenter la fiabilité. La section 3.1.2 présente les différents objectifs visés par les systèmes distribués, ainsi que les domaines associés à ces objectifs.

Toutes ces considérations sur les systèmes distribués, ainsi que de nombreux exemples de systèmes ou algorithmes distribués peuvent être trouvés dans [Tel91, Tel94, Lyn96].

3.1.1 Notions de base

Du fait que les systèmes distribués possèdent deux niveaux d'abstraction bien distincts (celui des processus et celui du système tout entier), on a besoin de notions relatives à ces deux niveaux.

On utilise la notion d'*état local*, ou tout simplement d'*état* pour désigner celui d'un processus. L'*état global* du système, c'est-à-dire la liste des états locaux des processus, est appelé *configuration*.

De même, pour caractériser l'évolution du système, on utilise le terme d'*action* pour désigner un pas d'évolution d'un processus, et le terme de *transition* pour un pas d'évolution du système global. Ainsi une transition correspond à ce qu'un (ou plusieurs) processus effectue(nt simultanément) une action.

3.1.2 Objectifs des systèmes distribués

Les situations impliquant les systèmes distribués sont très variées, et visent des buts différents. En voici les principaux :

- **L'échange d'informations :** Le besoin d'échanger des informations entre ordinateurs s'est développé dans les années soixante, alors que les grandes universités et les entreprises se dotaient de leurs propres serveurs. La coopération entre ces différentes organisations a été facilitée par l'échange de données entre ordinateurs, et elle a encouragé le développement de réseaux longue distance. Aujourd'hui ces réseaux sont partout, que ce soit à grande échelle (Internet reliant les différentes régions du monde) ou à plus petite échelle (Intranet reliant les différents ordinateurs d'une même entreprise). Ils permettent aux particuliers comme aux professionnels d'échanger des données, par exemple en téléchargeant des fichiers ou par courrier électronique.
- **Le partage des ressources :** Si les ordinateurs se sont répandus au point que de nombreuses personnes en sont équipées dans une entreprise, ce n'est pas le cas de tous les périphériques (imprimantes, unités de stockage,...) Il est alors judicieux de connecter des machines au sein d'un réseau local, pour leur permettre d'accéder à ces périphériques partagés.
- **Augmentation de la fiabilité :** Dans un système distribué, du fait de l'existence de petites entités (les nœuds du système), il est souhaitable que, alors que certains nœuds sont défaillants, d'autres continuent à fonctionner en prenant en charge le travail des nœuds défaillants. Ainsi, si l'on fait effectuer la même tâche par plusieurs processeurs distincts, en prenant en sortie le résultat le plus probable (*i.e.* obtenu par le plus grand nombre de processeurs), on augmente la fiabilité grâce à la répartition.
- **Augmentation des performances :** Le fait de disposer de plusieurs processeurs, que ce soit au sein d'une même machine ou *via* un réseau, peut permettre de partager une tâche à effectuer entre plusieurs processus, et de les faire tourner en parallèle. Ainsi le temps global nécessaire pour effectuer la tâche peut être diminué d'autant.

Dans le cadre de l'augmentation de la fiabilité et des performances, on considère surtout des machines multiprocesseurs. Il arrive également que des ordinateurs distants se mettent à participer à une même tâche, comme par exemple le projet RC5[1] qui consistait à trouver une clef de 64 bits en tentant toutes les solutions possibles. Ce projet a été mené à bien en près de 5 ans, et au total par plus de 300000 personnes. Les échanges d'information se font sur les réseaux, que ce soient les réseaux longue distance ou les réseaux locaux. Le partage des ressources se fait quant à lui principalement à l'échelle des réseaux locaux, mais pas uniquement.

3.1.3 Problèmes soulevés par les systèmes distribués

Ces systèmes distribués apportent d'une part des solutions nouvelles ou parfois plus rapides à de nombreux problèmes et soulèvent d'autre part de nouvelles questions. Les systèmes distribués ont besoin de méthodes particulières pour être étudiés car ils sont relativement différents des systèmes classiques.

Tout d'abord il est courant en algorithmique distribuée que chaque entité ne connaisse qu'une partie de l'état du système, contrairement au cas classique où l'on a une connaissance globale de celui-ci. Cette hypothèse est tout à fait raisonnable. En effet, on imagine mal, dans un système comportant de nombreux processus, que chacun d'entre eux maintienne une connaissance globale du système. Cela demanderait une place mémoire trop importante, ou beaucoup d'échanges de messages. Dans le cas distribué on doit donc réussir à accomplir des

[1] pour plus d'informations, regarder la page web : `http ://www.distributed.net/rc5/`

tâches en n'ayant qu'une connaissance partielle du système.
Lorsqu'on veut, dans le cas de l'exclusion mutuelle, s'assurer qu'un seul processus du réseau possède un certain privilège, la difficulté vient justement du fait qu'il est impossible, pour n'importe quel processus, de savoir si, à l'autre bout du réseau, un autre processus possède ou non un privilège.

Un autre point important est l'absence de temps global sur le système. Chaque processus autonome peut *a priori* évoluer selon une échelle de temps qui lui est propre. Ainsi on n'a pas vraiment de contrôle permettant de savoir dans quel ordre les processus vont agir. Certains événements peuvent même avoir lieu de manière simultanée. Ainsi, pour formaliser cette notion d'ordre ou de simultanéité des actions, on va avoir besoin d'un modèle, celui de *démon*, décrit à la section 3.5.1.

3.1.4 Quelques questions étudiées en algorithmique distribuée

Cette section présente une liste (évidemment non exhaustive) de problèmes soit ne se posant que dans le cadre distribué, soit pouvant être résolus de manière distribuée. Pour certains d'entre eux, des exemples d'algorithmes seront donnés dans les parties suivantes de cette thèse.

Élection

Dans certains algorithmes distribués, on a besoin d'avoir un processus distingué des autres, appelé *leader* dans la littérature anglaise, par exemple lorsqu'on a besoin d'un processus unique pour initialiser un algorithme, servir de référence à tous les autres, etc.
Le problème est, partant d'une situation où tous les processus sont dans le même état, d'arriver à une situation où un et un seul d'entre eux sait qu'il est leader, et tous les autres savent qu'il ne le sont pas.
On veut également que cet algorithme ne soit pas affecté si seulement une partie des processus participe à l'élection (c'est par exemple le cas quand les autres tombent en panne).
Dans certains cas, les processus sont indistinguables, c'est-à-dire qu'il n'ont pas de numéro ou d'identité pour les différencier, le réseau est dit *anonyme*. Il se pose alors un autre problème : le système peut se trouver dans une situation symétrique, de laquelle il lui est impossible de sortir avec un algorithme déterministe. Il faut alors faire appel à des algorithmes probabilistes.
On peut trouver des références sur les réseaux anonymes dans [Tel91], et tout un chapitre de [Tel94] est consacré aux algorithmes d'élection.

Consensus

Le problème du consensus est le suivant : on considère un système distribué, dans lequel certains processus peuvent tomber en panne et s'arrêter définitivement, et on souhaite cependant que les processus se mettent d'accord sur une certaine valeur.
On va alors concevoir un algorithme tel que :
- si tous les processus préfèrent initialement la même valeur, alors cette valeur va être choisie ;
- tous les processus qui ne tombent pas en panne vont finir par se décider ;
- tous les processus qui se décident choisissent la même valeur.

Un algorithme assurant le consensus doit alors être assez “à l’écoute” des valeurs préférées des processus pour assurer la première condition, mais ne doit pas se bloquer dans le cas où certains processus tomberaient en panne.

Exclusion mutuelle

Le problème de l’exclusion mutuelle, qui sera au centre de nombreux algorithmes présentés dans cette thèse, se pose dans le cadre suivant : plusieurs processus disposent d’une ressource partagée, qui ne peut être utilisée que par un seul d’entre eux à la fois, comme par exemple l’accès en écriture à un fichier ou à une imprimante partagée. On a alors besoin d’un algorithme distribué pour décider, lorsque plusieurs processus veulent simultanément accéder à la ressource, dans quel ordre ils vont le faire. Cela se fait en général au moyen d’un *jeton*, qu’un processus doit posséder pour accéder à la ressource, et qui va circuler dans le système.
Les protocoles d’exclusion mutuelle doivent satisfaire deux conditions :

Sûreté : à chaque instant, au plus un processus utilise la ressource ;

Vivacité : tout processus qui souhaite accéder à la ressource pourra le faire un jour.

Allocation de ressources

Ce problème est une généralisation de l’exclusion mutuelle présentée ci-dessus. En effet, on ne considère plus l’existence d’une seule ressource, mais de plusieurs. Celles-ci peuvent être partagées seulement par certains processus, et le conflit est à gérer entre ceux-ci.
On peut aussi envisager des algorithmes plus complexes où un processus, pour entrer dans sa section critique, a besoin d’un accès simultané à plusieurs ressources.

On définit alors un ensemble E de sous-ensembles de processus qui ne peuvent pas accéder aux ressources voulues tous en même temps. Si l’on reprend l’exemple de l’exclusion mutuelle, E est formé de tous les sous-ensembles contenant au moins deux processus.

L’algorithme d’allocation de ressources doit lui aussi vérifier deux conditions, qui peuvent être formulées de la façon suivante :

Sûreté : dans toute configuration accessible, le sous-ensemble de processus qui sont dans leur section critique n’appartient pas à E

Vivacité : tout processus qui souhaite accéder à sa section critique pourra le faire un jour.

Un exemple très connu d’algorithme d’allocation de ressources est celui du dîner des philosophes. Le chapitre 5 est entièrement consacré à sa version probabiliste, ainsi qu’à une variante dont nous avons prouvé la convergence sans besoin d’équité.

Arbres couvrants

Lorsqu’on travaille sur un graphe, on a parfois besoin que la topologie de ce graphe ait une forme particulière (arbre, anneau...). On doit alors extraire un sous-graphe du graphe original ayant la topologie voulue. Pour ce faire, on préfère en général utiliser des algorithmes distribués qui permettent, en parallélisant les opérations, d’être plus efficace, tant du point de vue du nombre de messages transmis que de celui du temps nécessaire pour construire le sous-graphe voulu.

Un exemple de ces sous-graphes est l’*arbre couvrant*. Dans un graphe connexe, on cherche parfois à trouver un arbre (donc un graphe sans cycle) qui contient tous les sommets. En effet, la topologie d’arbre est parfois préférée (voir section 3.2.2).

Détection de terminaison

Lorsqu'un algorithme non distribué termine, c'est généralement parce que le système se trouve dans un état terminal. Dans le cas d'un système distribué, il est fréquent que la terminaison soit implicite, c'est-à-dire que la configuration globale soit considérée comme terminale sans que les processus soient dans un état terminal.
On peut prendre l'exemple d'un algorithme distribué où les processus s'envoient des messages. Si à un certain moment il n'y a plus aucun message qui circule dans le réseau, et que tous les processus sont dans un état où ils attendent de recevoir un message, alors la configuration est terminale. Seulement chaque processus ne s'en rend pas compte directement.

Pour transformer un algorithme avec terminaison implicite en un algorithme avec terminaison explicite, on fait tourner deux algorithmes supplémentaires, un dit de détection de terminaison, et un autre d'annonce de terminaison, qui interagissent entre eux et avec l'algorithme original. Tout d'abord l'algorithme de détection de terminaison détecte que la configuration est terminale, puis appelle l'algorithme d'annonce qui va envoyer un message de terminaison à tous les processus.

Dans ce cas, l'algorithme de détection de terminaison doit satisfaire trois conditions :

Non-interférence : l'algorithme de détection de terminaison ne doit pas influer sur le déroulement de l'algorithme original.

Vivacité : si à un moment la configuration est terminale, alors l'algorithme d'annonce doit être appelé en un temps fini.

Sûreté : si l'algorithme d'annonce est lancé, alors la configuration doit être terminale.

Le livre [Tel94] présente un algorithme distribué simple nécessitant une détection de terminaison, ainsi que plusieurs algorithmes de détection de terminaison. L'un d'entre eux est présenté à la section 9.2.1 et la vivacité en est prouvée en utilisant la méthode décrite à la section 9.2.

Orientation

Le problème de l'orientation se pose sur un anneau. Initialement, chaque processus est "orienté", mais il n'y a pas de cohérence globale entre ces orientations. Le but est d'atteindre une situation où tous les processus sont orientés "dans le même sens", c'est-à-dire qu'on peut parcourir l'anneau en suivant le sens des orientations.
Les algorithmes d'orientation sont utilisés lorsque l'on veut ensuite appliquer un algorithme qui nécessite que chaque processus connaisse son successeur, comme c'est le cas par exemple dans certains algorithmes d'auto-stabilisation. Plus précisément, le principe est le suivant.

Chaque processus est relié à ses deux voisins, et étiquette initialement ses liens par *Succ* (successeur) et *Pred* (prédécesseur). Si à un instant ces deux étiquettes sont égales, le processus en change immédiatement une pour qu'il n'y ait plus de conflit. Initialement, il n'y a pas de cohérence de ces étiquettes sur tout le système.

Le but d'un algorithme d'orientation est, partant d'une telle configuration, d'arriver dans une nouvelle configuration où les étiquetages sont cohérents, à savoir que, pour toute arête (p, q), le processus p a étiqueté cette arête *Succ* si, et seulement si le processus q l'a étiquetée *Pred*.

L'article d'Israeli et Jalfon [IJ93] présente des algorithmes d'orientation, à la fois dans le cas déterministe et dans le cas probabiliste.

3.1.5 Tolérance aux fautes

Comme tout système, un système distribué est susceptible de subir des pannes (ou fautes), à savoir que tout ou partie du système peut, à un moment donné, cesser de réagir suivant sa spécification. Cette notion de faute recouvre des cas assez divers comme la corruption de code, les problèmes de transmission de messages, le ralentissement d'un processus, voire son arrêt intempestif.

Un système tolérant aux fautes devrait, idéalement, pouvoir continuer à fonctionner correctement dans de telles situations. Lorsque les fautes peuvent arriver n'importe quand et être de n'importe quelle nature, le problème devient vite très complexe.

Selon le type de faute possible, les façons d'y remédier sont diverses.
Si par exemple les fautes se situent au niveau de la transmission des messages, qui peuvent être légèrement déformés au cours de la transmission, il est possible d'utiliser des codes détecteurs d'erreur. Le principe est d'envoyer, en plus du message, une part d'information supplémentaire, qui est fonction du message. Le receveur peut alors récupérer les deux parties, recalculer la fonction, et si le résultat qu'il obtient n'est pas conforme à l'information supplémentaire envoyée, il détecte l'erreur et peut demander qu'on lui renvoie le message.
Si l'on veut pouvoir retrouver le message à partir du message modifié, on peut utiliser des codes correcteurs d'erreur. L'information est alors envoyée de manière redondante, si bien que, lorsque la quantité d'erreurs n'est pas trop grande, on peut reconstituer le message original directement à partir du message déformé.

Les messages peuvent aussi, suivant le modèle que l'on considère, être perdus, voire reçus plusieurs fois au lieu d'une, et il est parfois nécessaire de détecter ces problèmes. Par exemple si un ordre de débiter un compte en banque est reçu deux fois au lieu d'une, il ne faut pas que l'opération soit effectuée deux fois.

Les principaux modèles de pannes considérés sont les suivants :
- on dit d'une panne qu'elle est *transitoire* lorsqu'elle n'intervient qu'une seule fois et que, lorsqu'elle se termine, les processus reprennent une activité normale ;
- une panne est dite *définitive* (ou *fatale*) lorsqu'un (ou plusieurs) processus est(sont) définitivement arrêté(s) (on utilise le terme de *crash* en anglais) ;
- enfin les pannes *byzantines* sont celles qui sont totalement incontrôlables. Un processus peut s'arrêter de fonctionner, puis reprendre son activité.

Le qualificatif de "byzantin" vient d'un article de Lamport, Shostak et Pease [LSP82], présentant un problème de consensus sous la forme de discussions, par le biais de messagers, entre des généraux de l'armée byzantine, devant décider d'une éventuelle attaque commune contre une ville ennemie. La difficulté de ce problème vient du fait que certains des généraux peuvent être des traîtres, et donc agir à leur guise, envoyant des mauvaises informations, voire des informations contradictoires.

Selon le type de panne, il faut concevoir des algorithmes soit pour les gérer, et donc continuer à fonctionner correctement même pendant la panne (sous réserve que celle-ci ne soit pas trop étendue), soit pour s'en remettre, à savoir permettre au système, après la panne, de revenir à une situation et un comportement corrects.

En ce qui concerne cette deuxième possibilité, et lorsque le retour à un comportement normal se fait sans aide extérieure, on parle d'auto-stabilisation. Cette notion est présentée à la section 3.6.4, et plusieurs exemples d'algorithmes auto-stabilisants seront étudiés au cours de cette thèse, comme par exemple l'algorithme d'Herman d'exclusion mutuelle auto-stabilisante [Her90] décrit puis prouvé aux exemples 3.20, page 62 et 4.3, page 69.

3.2 Le modèle

Après cette introduction portant principalement sur les enjeux et les avantages de l'algorithmique distribuée, nous allons revenir sur le modèle, en décrivant plus précisément comment fonctionne un tel système. Tout d'abord, du fait que les systèmes distribués se composent de plusieurs entités, il faut préciser la façon dont elles communiquent entre elles (voir section 3.2.1). Ensuite, on peut s'intéresser à l'agencement de ces entités, savoir qui peut communiquer avec qui, et définir ainsi la topologie du réseau sous-jacent (section 3.2.2). Enfin il est intéressant de préciser la façon dont on va représenter les systèmes (section 3.2.3) ainsi que les transitions (section 3.2.4).

3.2.1 Modes de communication

Pour étudier un système distribué, il faut prendre en compte la façon dont les différents nœuds communiquent entre eux, et donc envisager deux possibilités : l'échange de messages, et les variables partagées.

Communication par échange de messages

Dans ce cas, on considère qu'il existe des canaux de communication entre les différents nœuds, par lesquels peuvent passer des informations. Chaque processus peut donc effectuer, outre des actions internes, deux actions particulières : l'envoi et la réception de message.
L'échange de messages est dit *synchrone* lorsque tout envoi de message se synchronise avec la réception correspondante pour ne plus former qu'une seule transition du système. Ainsi un processus ne peut envoyer un message que si le processus correspondant est prêt à le recevoir. Lorsque les messages peuvent rester un temps non nul dans le canal de communication, on parle de communication *asynchrone*. Le message est alors stocké dans le canal jusqu'à ce que le processus correspondant soit prêt à le recevoir.

Dans certains cas, comme par exemple l'auto-stabilisation, présentée à la section 3.6.4, les communications par échange de messages ne sont pas appropriées, car si l'on considère le modèle de communication asynchrone, le système peut être bloqué dans une configuration où chaque processus attend de recevoir un message et où tous les canaux sont vides (voir la détection de terminaison, section 3.1.4). En général, toutes ces configurations ne sont pas *légitimes*, et donc le système n'est pas auto-stabilisant.

Communication par variables partagées

Le principe de base de la communication par variables partagées est le suivant : un processus possède une variable, qui ne peut être modifiée que par lui, mais qu'il partage en lecture avec un ou plusieurs autres processus. Par exemple, dans le cas d'automates, on peut considérer que chaque automate peut lire l'état de ses voisins et changer son propre état en conséquence. Pour ce qui est de la lecture des variables partagées, on peut considérer deux modèles différents : *read one* pour lequel un processus ne peut lire que l'état d'un de ses voisins en une action, et *read all* pour lequel la lecture des états de tous les voisins se fait en une seule étape.

Il arrive de considérer des systèmes un peu plus généraux, où il existe des variables partagées qui peuvent être lues et modifiées par un certain sous-ensemble de processus. Dans le cas de l'algorithme d'exclusion mutuelle d'Israeli et Jalfon [IJ90], chaque processus accède en lecture et en écriture à son propre état ainsi qu'à ceux de ses deux voisins immédiats (voir l'exemple 4.2.1, page 72).

Dans la suite de la thèse, nous allons nous restreindre à des systèmes communiquant par variables partagées.

3.2.2 Topologie des systèmes distribués

Dans un système distribué, savoir quel processus peut communiquer avec quel autre est très important. En effet, un système en étoile, où chaque élément est relié à un même nœud central, n'aura pas le même comportement par exemple qu'un système où les éléments sont reliés en anneau. C'est pourquoi cette section va présenter quelques topologies particulières de graphes sous-jacents aux systèmes distribués.

Pour chaque processus, on définit l'ensemble de ses *voisins* par l'ensemble des processus desquels il peut recevoir des informations, soit, dans le cas des variables partagées, l'ensemble des processus dont il peut lire une variable partagée.
Un système est dit *unidirectionnel* s'il existe deux processus p et q dans le système tels que p peut transmettre des informations à q mais l'inverse est faux. Un système est *bidirectionnel* si, dès lors que p peut transmettre des informations à q, alors q peut également transmettre des informations à p.

La topologie d'un système distribué peut être représentée par un graphe $G = (S, A)$ où l'ensemble S des sommets de G est égal à l'ensemble des processus du système, et où les arêtes de A symbolisent la communication entre ces processus. Lorsque le système est unidirectionnel, on le représente par un graphe orienté (les arêtes ont un sens). Lorsque le système est bidirectionnel, un graphe non orienté suffit.

Un graphe non orienté G est dit connexe lorsque, pour tout couple de sommets (s, s'), il existe un chemin dans le graphe reliant ces deux sommets, plus précisément il existe des sommets $s = s_1, s_2, ..., s_k = s'$ tels que pour tout indice $1 \leq i < k$, l'arête (s_i, s_{i+1}) (et donc (s_{i+1}, s_i)) est dans G.

On peut à présent donner quelques topologies particulières :

Anneau : Un anneau à N sommets est un graphe pour lequel il existe une numérotation des sommets de s_0 à s_{N-1}, telle que les arêtes sont du type (s_i, s_{i+1}) ou (s_{i+1}, s_i) (les indices sont pris modulo N). Cette forme de réseau est souvent utilisée du fait de sa simplicité, par exemple pour des algorithmes d'exclusion mutuelle auto-stabilisante (voir section 3.6.4). Avec cette topologie, tout processus communique avec au plus deux voisins.

Arbre : Un arbre à N nœuds est un graphe connexe à $N-1$ arêtes. Cette définition implique qu'un arbre ne contient pas de cycle. Lorsqu'il est bidirectionnel, un arbre assure qu'entre deux sommets, il y a exactement un chemin. Les topologies en arbre sont utilisées dans le cadre distribué car elles possèdent un nombre d'arêtes minimal, qui peut permettre de minimiser les coûts de communication. Il existe également certains problèmes qui peuvent être résolus plus efficacement sur les arbres, comme par exemple le cas des algorithmes par vagues (*wave algorithms* en anglais) qui permettent entre autres la diffusion d'information avec retour, ou la synchronisation au sein d'un réseau (voir [Tel94] pour des exemples d'algorithmes).

Étoile : Un réseau est dit en étoile s'il possède un nœud central, et $N-1$ arêtes connectant les $N-1$ autres nœuds au centre. Cette topologie permet de représenter des réseaux où un processus joue le rôle de serveur pour tous les autres. Le problème des réseaux en étoile est que toutes les communications passent par le nœud central, au risque de ralentir le réseau, voir de le paralyser si le serveur est défaillant.

3.2.3 Systèmes de transitions

Du fait qu'il peut se décrire par un ensemble de configurations et des règles qui précisent dans quel cas on peut passer de l'une à l'autre, un système distribué peut être vu comme un système de transitions.

Dans ce cas, on le représente par un graphe, qu'il ne faut pas confondre avec celui présenté à la figure 3.1. Dans cette figure le graphe représentait la topologie du réseau, avec un nœud pour chaque processus et des arêtes pour représenter la communication. Ici, un nœud représente une configuration, c'est-à-dire la description de l'état de chaque processus, et une arête entre deux configurations signifie que l'on peut passer de l'une à l'autre.

La section suivante présente une façon de décrire les actions possibles dans le système.

3.2.4 Description des actions

De façon générale, une transition d'un système distribué peut se décrire sous forme (d'un ensemble) de commandes gardées du genre `SI` la garde est vraie `ALORS` on effectue l'action correspondante :

```
SI Pi est dans l'état q
et Pj est dans l'état q1
et Pk est dans l'état q2
ALORS Pi passe dans l'état q3
```

Les processus `Pj` et `Pk` sont ici les voisins du processus `Pi`.

Ce type de règle implique que la lecture des états des voisins se fait de manière simultanée et qu'on est sous l'hypothèse de *read all* décrite à la section 3.2.1.

Si on revient au modèle des systèmes de transitions décrit à la section précédente, alors, pour représenter cette seule commande gardée, il faut, dans le système de transitions, mettre une arête entre tout couple de configurations (x, x') tel que :

– la configuration x vérifie la garde de la commande ci-dessus ;
– la configuration x' est semblable à x sauf pour l'état du processus `Pi` qui est `q3`.

La représentation d'un système distribué par un système de transitions n'est donc en général pas adaptée, car le nombre de sommets est de l'ordre de nbE^N où nbE vaut le nombre d'états maximum possible pour chaque processus, et N vaut le nombre de processus dans le système. Elle se justifie par contre pour illustrer certaines notions définies sur les systèmes de transitions, et qui s'appliquent donc aux systèmes distribués, comme par exemple la convergence et l'auto-stabilisation (voir la figure 3.4, page 58).

Lorsque plusieurs actions sont effectuées en même temps, c'est le même principe : premièrement toutes les lectures se font simultanément, et ensuite seulement, les états de certains processus sont modifiés.

À ce formalisme de commandes gardées, on va dans la suite préférer décrire ces transitions à l'aide de règles de réécriture (décrites à la section 3.4), en utilisant le fait que les systèmes que nous allons considérer ont une topologie en anneau (voir section 3.3). Cependant, pour coller à la description de ses concepteurs, l'algorithme de Kakugawa et Yamashita sera présenté sous forme de commandes gardées à la section 4.2.3.

3.3 Systèmes distribués en anneau

Dans la suite de cette thèse, nous allons nous intéresser au cas particulier où les différents processus sont disposés en anneau. On dispose donc d'un nombre paramétré N de machines (voir section 3.7 pour les problèmes soulevés par l'aspect paramétré), par exemple N automates finis, et pour effectuer une transition, chaque automate peut prendre en compte, outre son état propre, celui de son (ses) voisin(s) immédiat(s). L'état de chaque processus est donc considéré comme une variable partagée, sur laquelle seul le processus lui-même possède le droit d'accès en écriture, mais son(ses) voisin(s) possèdent un droit d'accès en lecture.

Cet exemple est illustré dans la figure 3.2, où une flèche d'un processus P vers un processus Q signifie que Q peut lire l'état de son voisin P. Dans ce premier exemple, on aurait pu ne pas représenter les flèches sur les transitions, du fait que le réseau est bidirectionnel.

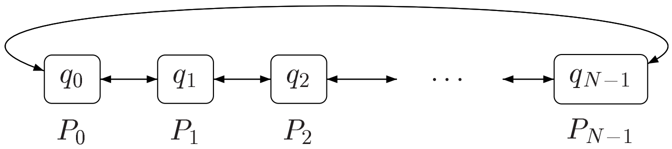

FIG. 3.2 – Système en anneau où chaque processus lit l'état de ses deux voisins.

On peut aussi imaginer le cas d'un réseau unidirectionnel où chaque processus ne peut lire que l'état de son voisin de gauche (figure 3.3). Selon la définition de voisin donnée à la section 3.2.2, le processus à sa droite n'est donc pas un voisin.

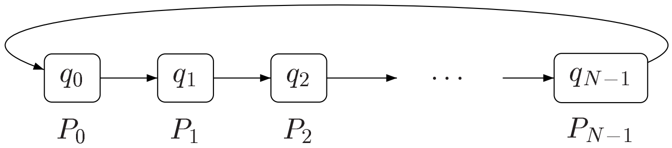

FIG. 3.3 – Système en anneau où chaque processus peut lire l'état de son voisin de gauche.

Nous considérerons aussi le cas où les processus aux positions 0 et $N-1$ ne communiquent pas. On parle alors de *topologie linéaire* ou de chaîne de processus.

3.4 Systèmes distribués et réécriture

Lorsque la topologie du système est un anneau ou une chaîne, on peut décrire une configuration (*i.e.* état global du système) non plus comme une suite d'états locaux, mais comme un *mot*, à savoir une suite de lettres de Σ. On va donc considérer l'ensemble Σ des états possibles pour chaque processus comme un alphabet. Par exemple, si on écrit $P_i = q_i$ pour dire que le processus P_i est dans l'état q_i, on a utiliser la notation suivante :

$$P_0 = q_0, P_1 = q_1, ..., P_{N-1} = q_{N-1} \longleftrightarrow q_0q_1...q_{N-1}$$

qui assimile une configuration du système à N processus et un mot de Σ^N. L'ensemble Σ^* des configurations est également noté X. Dans une telle configuration, on numérote les N processus de 0 à $N-1$, ainsi on peut raisonner sur les *indices* ou *positions* des processus modulo N. Dans le cas où l'on s'intéresse à un anneau, les processus d'indice $N-1$ et 0 communiquent. Dans le cas d'une chaîne, non.

Comme une configuration correspond à un mot, et réciproquement, on peut utiliser indifféremment les termes de *motif* ou de *facteur* pour désigner une suite d'états de processus consécutifs. Par exemple, la configuration $x = abcbca$ contient le motif bcb, autrement dit bcb est un facteur de x.

En utilisant cette représentation d'une configuration par un mot, on va pouvoir utiliser des notions de réécriture, et plus particulièrement la notion de règle.

Définition 3.1. *Un* système de réécriture $\mathcal{S}$ *est un sous-ensemble de* $\Sigma^* \times \Sigma^*$ *dont les éléments, appelés* règles de réécriture*, s'écrivent sous la forme* $u \to v$ *où* u *et* v *sont des mots de* Σ^**. Le mot* u *est appelé* membre gauche *de la règle et* v *en est le* membre droit.

L'application d'une règle de réécriture se fait de la manière suivante : soit $R_1 = u \to v$ une règle de réécriture, et soit $t = t_1 u t_2 \in \Sigma^*$ un mot. Comme t contient le facteur u, on peut appliquer la règle R_1 à t pour donner $t' = t_1 v t_2$. On dit alors que t se *réécrit* en t' *via* la règle R_1.

Dans certains systèmes distribués, il arrive que les processus ne soient pas tous identiques, et donc n'exécutent pas tous le même code. Cela se traduit ici par le fait que des processus d'indices différents aient un ensemble de règles différent.

Quelques algorithmes de ce type ([Gho93, BD95]) seront présentés dans la partie III, distinguant les deux processus des extrémités, qui ont des ensembles de règles particuliers appelés *Top* et *Bottom* (voir section 7.1).

Comme on va s'intéresser à des systèmes pour lesquels le nombre N de processus est paramétré (voir section 3.7) mais constant le long d'une suite de transitions, on ne va considérer que des règles de réécriture qui préservent la longueur des mots (*i.e.* $|u| = |v|$).

Une transition consiste alors, dans notre système, en l'application d'une règle de réécriture à une certaine position. Dans certains des cas que l'on considérera dans la suite, ces règles ne changent que l'état d'un processus. Dans ce cas on va considérer que la position à laquelle on applique la règle est celle du processus dont l'état est modifié. Lorsque plusieurs états seront changés, on définira le moment venu notre convention sur le processus auquel la règle est appliquée.

Définition 3.2. *Étant donnée une configuration* x*, une position est dite* activable *s'il est possible, dans* x*, d'appliquer une règle à cette position. On note* $\mathcal{E}(x)$ *l'ensemble des processus activables de la configuration* x*.*

Par abus de langage, on dira d'un processus qu'il est activable dans x s'il est à une position activable.

Exemple 3.3. Considérons un système distribué assez simple :
- un anneau de 5 processus,
- un ensemble d'états $\Sigma = \{0, 1\}$ et
- une seule règle de réécriture :

$$01 \to 00$$

La configuration $x = 10110$ possède deux positions activables : 0 et 2 (car le "voisin de gauche" du processus à la position 0 est le processus à la position $0 - 1$ modulo $5 = 4$). À partir de x il y a *a priori* trois transitions possibles : si l'on applique la règle de réécriture à la position 0 on obtient la transition $x \to x_1 = 00110$, on peut également appliquer la règle à la position 2 et obtenir $x \to x_2 = 10010$, ou même simultanément à ces deux positions et obtenir la transition $x \to x_3 = 00010$.

La configuration x_3 ne possède alors plus qu'une seule position activable qui est 3, et si on applique la règle encore une fois on arrive dans la configuration $x_4 = 00000$ qui n'a plus de position activable.

Le système décrit ci-dessus réalise la tâche suivante : si initialement la configuration contient au moins un processus dans l'état 0, alors en un nombre fini d'étapes on n'aura plus que des 0 dans la configuration.
On remarque qu'avec ce système il ne peut pas y avoir un nombre infini de transitions successives. En effet, toute application de la règle fait disparaître un 1, et ils sont initialement en nombre fini. ♣

3.5 Démons

3.5.1 Notion de démon

Pour décrire le comportement d'un système, il ne suffit pas de connaître les règles qui définissent les actions des processus : il faut également savoir dans quel ordre ces actions vont être effectuées. Une transition est donc caractérisée par deux choses : tout d'abord le choix des processus activables qui vont effectuer une action, et ensuite les règles qui sont appliquées à ces processus.

Ainsi, pour être précis, une transition de la configuration u vers la configuration v, en appliquant les règles $R_1, ..., R_k$ aux processus d'indices $i_1, ..., i_k$ doit être notée

$$u \xrightarrow[R_1,...,R_k]{i_1,...,i_k} v.$$

Remarque : La notation ci-dessus peut laisser croire que l'on peut appliquer simultanément des règles à n'importe quelles positions activables mais ce n'est pas le cas. En effet, on ne peut bien évidemment pas autoriser d'appliquer simultanément deux actions qui modifient l'état du même processus. La remarque de la section 3.5.2 reviendra sur cette restriction.

Dans la pratique, on omettra souvent l'ensemble de règles. En effet, sur tous les systèmes que l'on va considérer, les membres gauches des règles sont mutuellement exclusifs. Cela signifie que, une fois que l'on a choisi la position à laquelle appliquer une règle, alors une seule d'entre elles est applicable.

Dans le même esprit, on va définir la notion d'exécution dans ce contexte. Dans un système distribué, l'équivalent des actions de la section 2.3 est le choix de processus activables fait par le démon. Ainsi, une *exécution* (resp. une *exécution finie*) est une suite infinie (resp. finie) de transitions consécutives.

Remarque : Même s'il est dit plus haut que, une fois les indices de processus choisis, le choix des règles s'impose de lui-même, il faut garder à l'esprit que le démon n'a pas un contrôle total sur le système. En effet, dans les systèmes distribués probabilistes (voir section 3.8), choisir une règle n'implique pas choisir son membre droit, qui est tiré aléatoirement selon une certaine distribution de probabilités.

Pour décrire la notion d'ordonnancement des actions, on va considérer un mécanisme extérieur, que l'on appelle *démon*, qui n'est autre qu'un *ordonnanceur* (*scheduler* en anglais) qui va choisir l'ordre suivant lequel les processus vont effectuer des actions. Il sélectionne, pour chaque étape (ou transition), le ou les processus qui vont effectuer une action.

Bien évidemment, le démon ne peut pas choisir à chaque étape n'importe quels processus : il va les choisir parmi ceux qui sont *activables*, ceux qui peuvent effectuer une action. Plus précisément :

Définition 3.4. *On appelle* démon *tout mécanisme qui, à chaque étape d'une exécution, choisit un sous-ensemble non vide de processus activables, auxquels on va appliquer une règle.*

Le choix du terme “mécanisme” est volontaire. En effet, on ne se préoccupe pas de savoir (ce) qui choisit les processus, mais uniquement de ce choix. Celui-ci peut se faire de manière déterministe, probabiliste, ou même purement déterministe suivant le démon considéré. De même, le mécanisme de choix peut prendre en compte tout l'historique de l'exécution ou uniquement l'état courant. Le démon peut également *a priori* choisir un ou plusieurs processus parmi ceux qui sont activables. Le démon peut donc être vu comme un mécanisme qui va restreindre l'ensemble des exécutions possibles.

Cette définition de démon est donc peu restrictive, et permet de représenter une gamme assez large de comportements. On peut ainsi représenter des cas très divers, comme par exemple un système où l'ordonnancement est fixé par avance, et le démon n'a aucune liberté (il doit suivre cet ordonnancement), le cas où le démon choisit aléatoirement parmi les processus activables, ou encore le cas où le démon est totalement libre de ses choix et choisit totalement arbitrairement des processus parmi ceux qui sont activables.

Exemple 3.5. Avec cette définition, on peut avoir une très grande variété de démons. Par exemple, considérons le cas d'un algorithme où les processus ont deux états, actif (ils essaient d'effectuer une certaine tâche) et passif. avec les deux règles associées : $a \to p$ et $p \to a$.
Supposons de plus qu'il existe une certaine limitation dans le système disant que dès que plus de trois processus sont actifs en même temps, alors ils ne parviennent pas à effectuer leur tâche et repassent tous dans l'état passif.
Ceci peut être représenté par un démon qui, tant qu'il n'y a pas plus de trois processus actifs en même temps, sélectionne un ou plusieurs processus pour les faire changer d'état, et dès que plus de trois processus sont actifs en même temps, alors il sélectionne tous les processus actifs, et ceux-ci vont donc repasser dans l'état passif. ♣

Selon les cas, ce démon peut aussi être appelé *adversaire*, par exemple lorsque, par ses choix à chaque transition, il essaie d'empêcher le système d'effectuer la tâche pour laquelle le système est conçu.

3.5.2 Caractéristiques des démons

Selon les propriétés que l'on souhaite que le système vérifie, et l'environnement dans lequel ce système évolue, on va considérer différents types de démons, et donc d'ordonnancement :

- Un ordonnancement est dit *centralisé* si, à chaque étape, un seul processus est sélectionné. Un démon est dit *centralisé* si tous ses ordonnancements possibles sont centralisés. Dans le cas contraire il est appelé démon *distribué.*
- Un démon est *synchrone* si, à chaque étape, il sélectionne tous les processus activables (c'est-à-dire $\mathcal{E}(x)$ tout entier). Dans ce cas il a un seul ordonnancement possible appelé *ordonnancement synchrone*

Le modèle de démon centralisé est utilisé par exemple lorsque l'on veut considérer un système totalement asynchrone, où il n'y a strictement aucune chance que deux processus effectuent une action exactement en même temps.

Remarque : Le fait de considérer un démon distribué, qui peut sélectionner plusieurs processus en même temps, peut poser problème lorsque l'on considère des actions (règles) qui peuvent modifier plusieurs lettres à la fois. En effet si l'on applique simultanément deux règles à deux positions proches, et qu'elles sont censées modifier l'état d'un même processus, il y a conflit.
Dans le cas où les règles modifient l'état de plusieurs processus, on est alors obligé de restreindre les possibilités du démon, pour éviter ce genre de conflit. Par contre, dans le cas où les règles ne réécrivent qu'une seule lettre, il n'y a pas de problème.

3.5.3 La propriété d'équité

Il existe, selon le contexte, une multitude de propriétés désignées par le terme d'équité. Elles ont en commun l'idée selon laquelle on ne veut pas qu'un participant, une action ou un processus soit défavorisé par rapport à un autre. Ici nous considérerons la notion d'équité sur les processus.

Définition 3.6. *Une exécution est dite* faiblement équitable *si, le long de cette exécution, il n'existe pas de processus qui soit activable dans une infinité de configurations consécutives mais qui n'effectue jamais une action.*
Une exécution est dite fortement équitable *si, le long de cette exécution, il n'existe pas de processus qui soit infiniment souvent activable et qui n'effectue jamais une action.*

La première notion d'équité faible est parfois appelée *justice.* Dans le cas de l'équité forte, le terme de *compassion* est également employé (voir par exemple [KPSZ02]).

On peut remarquer que ces définitions sont équivalentes aux caractérisations suivantes :

Proposition 3.7. *Une exécution est faiblement équitable si et seulement si, le long de cette exécution, tout processus qui est activable dans une infinité de configurations consécutives effectue infiniment souvent une action. Une exécution est fortement équitable si et seulement si, le long de cette exécution, tout processus qui est infiniment souvent activable effectue infiniment souvent une action.*

Démonstration. La preuve se fait par induction. D'après la définition, pour une exécution faiblement équitable, chaque processus (prenons-en un au hasard, et appelons le P) qui est activable dans une infinité de configurations consécutives effectue au moins une fois une action, par exemple dans la $i^{\text{ème}}$ transition de l'exécution. Si l'on coupe le début de l'exécution en ne gardant que ce qu'il se passe à partir de la $i+1^{\text{ème}}$ configuration, on a toujours une exécution (infinie), le long de laquelle P est activable dans une infinité de configurations consécutives. Il

existe donc un indice $j > i$ tel que P effectue une action dans la $j^{\text{ème}}$ transition de l'exécution d'origine. En itérant cet argument, on en déduit que le processus P effectue un nombre infini d'actions.
Pour ce qui est de l'équité forte, le raisonnement est exactement le même. □

Exemple 3.8. Considérons un système à deux processus, numérotés 0 et 1, et une exécution le long de laquelle les processus activables sont $\{0,1\},\{1\},\{0,1\},\{1\},...$. Celle-ci est faiblement équitable si le processus 1 est infiniment souvent sélectionné, et elle est fortement équitable si les deux processus sont infiniment souvent sélectionnés. ♣

Lorsque, comme dans le cas de l'algorithme du dîner des philosophes probabilistes de Lehmann et Rabin présenté au chapitre 5, à tout moment, chaque processus est activable, les deux conditions coïncident, et on obtient la propriété suivante.
Une exécution est équitable si, le long de cette exécution, chaque processus est infiniment souvent sélectionné par le démon (et donc effectue infiniment souvent une action).

On peut étendre cette notion aux démons ou aux ordonnancement, en disant qu'un démon (ou un ordonnancement) est équitable si toutes les exécutions qu'il engendre le sont, c'est-à-dire si toute suite infinie de choix successifs possibles pour ce démon génère une exécution équitable.

Selon les systèmes, le statut de la notion d'équité est différent. Dans certains systèmes, les processus effectuent régulièrement des actions, et l'équité est une propriété de base du système. Par exemple pour l'algorithme original des "free philosophers" de Lehmann et Rabin ([LR81], voir aussi section 5.2), il faut supposer que le système est équitable pour vérifier une propriété de convergence.
Dans d'autres contextes, il arrive que l'on ne suppose pas l'équité du démon, mais qu'au contraire on veuille la démontrer. On prend donc un démon *a priori* arbitraire (parfois on se restreint aux démons centralisés quelconques), et on cherche à prouver que toute exécution sera forcément équitable. Dans ce cas, lorsque tous les démons possibles pour cet algorithme sont équitables, on parle d'*algorithme équitable*. C'est le cas de l'algorithme de Beauquier, Gradinariu et Johnen présenté à la section 4.2.2 où les auteurs introduisent une notion de jeton déterministe, qui va permettre d'assurer l'équité du système. C'est aussi le cas des algorithmes de Ghosh et de Beauquier-Debas ([Gho93, BD95]) qui seront présentés dans la partie III, et où nous utiliserons le fait que l'algorithme est équitable pour effectuer notre preuve de convergence (voir chapitre 8).

3.6 Vérification de systèmes distribués

Dans cette dernière partie du chapitre sur les systèmes distribués, nous donnons un point de vue sur la vérification, que ce soit dans le cadre général ou celui, plus particulier, des systèmes distribués.
La section 3.6.1 mentionne les logiques temporelles dans le cadre desquelles vont rentrer les propriétés que nous allons vérifier. Nous donnons ensuite à la section 3.6.2 deux types bien particuliers de propriétés : celles de sûreté et celles de vivacité. Pour finir, nous nous intéressons plus précisément aux propriétés que nous allons étudier : la convergence à la section 3.6.3 et l'auto-stabilisation à la section 3.6.4.

3.6.1 Logiques temporelles

Dans toute la suite de cette thèse, nous allons nous intéresser à prouver des propriétés sur les systèmes distribués. Le côté arborescent de nos systèmes (dans une configuration, plusieurs transitions sont possibles, et donc plusieurs états successeurs) nous amène à considérer des logiques de temps arborescent, comme CTL et CTL*. Plus précisément, les propriétés que nous allons considérer entrent dans le cas de la logique CTL car elles sont de la forme :

"Pour tout chemin, on arrivera un jour dans un état légitime"

Du fait de la simplicité de l'énoncé des propriétés que nous allons chercher à prouver, nous allons les décrire en langage courant. C'est pourquoi nous ne donnerons pas ici la syntaxe et la sémantique précise de ces logiques, qui peuvent être trouvées dans [Eme90].

Dans le cadre probabiliste, nous utilisons des propriétés exprimables dans la logique temporelle probabilistes pCTL. Cette logique est présentée à la section 3.8.1, pour permettre d'énoncer le résultat de Bianco et de Alfaro annonçant qu'on peut restreindre l'ensemble des ordonnancements considéré pour vérifier des propriétés de pCTL.

3.6.2 Classification des propriétés

Parmi les propriétés temporelles que l'on va vouloir vérifier sur un système, on peut distinguer deux types particuliers : les propriétés de sûreté et les propriétés de vivacité. Cette classification, ainsi que d'autres classes (non disjointes) de propriétés, sont très bien décrites dans le livre [MP92] de Manna et Pnueli, dans le cadre de la logique temporelle linéaire (LTL).

Sûreté

Les propriétés de sûreté sont utilisées dans le cas où l'on veut montrer qu'un certain "mauvais" comportement n'arrive jamais.
Ce genre de vérification se fait lorsqu'on veut détecter des erreurs sur des systèmes critiques, s'assurer qu'une situation aberrante ne peut arriver.

Exemple 3.9. Si l'on considère le cas d'un distributeur de billets, on peut vouloir s'assurer par exemple que, lorsque le distributeur est vide, il ne va plus essayer de distribuer de billets (et surtout qu'il ne va pas donner l'ordre d'enlever ce montant du compte en banque correspondant).

Cela s'exprime par le fait que l'état où le distributeur est vide et où il essaie de donner des billets est inaccessible. ♣

L'exemple ci-dessus, même s'il ne correspond pas à un système distribué, est un bon exemple de propriété de sûreté.

Vivacité

Le deuxième type de propriétés s'intéresse à une question totalement différente : informellement, savoir si un "bon" événement va forcément arriver. Ce genre de propriété permet de tester la réactivité d'un système.

Exemple 3.10. Si l'on reprend l'exemple du distributeur de billets, on peut par exemple vouloir s'assurer que le distributeur va toujours revenir dans un état d'attente, où il a fini une transaction et attend la carte de la personne suivante. ♣

Caractérisation de ces propriétés

On dispose d'une caractérisation formelle des propriétés de sûreté et de vivacité :

- Une formule φ définit une propriété de sûreté si et seulement si, toute suite d'états $\rho = x_0, x_1, ..., x_n, ...$ ne satisfaisant pas φ (donc satisfaisant $\neg\varphi$) possède un préfixe fini $x_0, x_1, ..., x_k$ dont aucune extension infinie ne satisfait φ.
- Une formule φ définit une propriété de vivacité si, et seulement si, toute suite finie d'états $x_0, x_1, ..., x_k$ peut être étendue à une suite infinie qui satisfait φ.

Remarque : Les caractérisations précédentes parlent de “suites d'états”, et non pas d'exécutions. Ainsi, pour les propriétés de vivacité, l'extension de la suite d'états ne correspond pas forcément à une exécution. Si c'était toujours le cas, cela signifierait que, pour tout système distribué et toute propriété de vivacité, il existe une exécution du système qui satisfait la propriété. Et cela n'est pas vrai en général. Par exemple si le système comporte un blocage, certaines exécutions finies ne seront pas prolongeables, ne pourront donc pas être étendues, et ne satisferont pas les propriétés de vivacité.

Dans leur livre, Manna et Pnueli s'intéressent à LTL (Linear Temporal Logic), et expriment donc uniquement des propriétés sur un chemin. Dans notre étude, nous allons aussi nous intéresser à des quantifications sur les chemins, et donc plutôt utiliser des propriétés exprimables en CTL (comme annoncé à la section 3.6.1), ou son équivalent probabiliste pCTL (décrit à la section 3.8.1).

Méthodes classiques de vérification de propriétés de sûreté

Dans cette thèse, nous allons nous intéresser à la vérification de propriétés de vivacité, et plus précisément de convergence. Il est cependant intéressant de rappeler comment peut se faire la vérification de propriétés de sûreté.

L'approche classique est d'identifier d'un côté les états initiaux, de l'autre les “mauvais états” (ceux que l'on veut que le système n'atteigne pas) et, par un calcul d'accessibilité, de vérifier que ces mauvais états ne peuvent pas être atteints à partir des états initiaux (analyse en avant), ou que les états initiaux ne font pas partie des états qui mènent à l'ensemble des mauvais états (analyse en arrière).

Comme ce type d'analyse ne termine pas toujours lorsque les systèmes sont infinis ou paramétrés, de nombreux travaux utilisent des méthodes d'accélération, pour calculer d'un coup des états accessibles en plus d'une étape dans le système original (*cf.* section 3.7).

3.6.3 Propriété de convergence

Dans le reste de cette thèse, nous allons nous intéresser à un cas particulier de propriétés de vivacité : la *convergence*. Elle est exprimé ici dans le cadre plus général des systèmes de transitions.

Définition 3.11. *On dit qu'un système de transitions $\mathcal{S}$* converge *vers un ensemble $\mathcal{L}$ de configurations si,*

- *il existe une transition partant de tout état n'appartenant pas à $\mathcal{L}$;*
- *pour tout état x_0, pour toute exécution $x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_n \rightarrow ...$ de $\mathcal{S}$, il existe $i \in \mathbb{N}$ tel que la configuration x_i soit dans l'ensemble $\mathcal{L}$.*

Cette définition correspond bien à la notion de convergence : quel que soit l'endroit d'où l'on parte, tant que l'on n'est pas dans $\mathcal{L}$, on peut prolonger l'exécution, et comme on ne peut pas rester infiniment hors de $\mathcal{L}$, on va forcément atteindre un état de $\mathcal{L}$.
Dans le cas déterministe, on veut considérer tous les choix (non déterministes) possibles du démon et montrer que, quels qu'ils soient, on va arriver dans l'ensemble $\mathcal{L}$. Dans le cas probabiliste, il faut ici fixer un ordonnancement et montrer que la convergence à lieu sous cet ordonnancement avec probabilité 1. On peut ensuite vouloir montrer que, quel que soit, l'ordonnancement, on a toujours convergence avec probabilité 1 (voir section 3.8).

Lorsque l'on n'impose pas que tout état hors de $\mathcal{L}$ est réductible, on a la notion (plus faible) d'ensemble *inévitable* :

Définition 3.12. *Soit $\mathcal{S}$ un système de transitions. On dit qu'un ensemble de configurations $\mathcal{L}$ est* inévitable *pour $\mathcal{S}$ si pour tout état x_0, pour toute exécution (infinie) $x_0 \to x_1 \to ... \to x_n \to ...$ de $\mathcal{S}$, il existe $i \in \mathbb{N}$ tel que la configuration x_i soit dans l'ensemble $\mathcal{L}$.*

Dans la définition de convergence, il est dit que toute exécution passe au moins une fois par l'ensemble $\mathcal{L}$, mais en fait elle est équivalente à la propriété suivante :

Proposition 3.13. *Soit $\mathcal{S}$ un système de transitions qui converge vers un sous-ensemble $\mathcal{L}$ de configurations. Pour tout état x_0, pour toute exécution $x_0 \to x_1 \to ... \to x_n \to ...$, il existe une infinité de $i_k \in \mathbb{N}$ distincts ($k \in \mathbb{N}$) tels que toutes les configurations x_{i_k} pour $k \in \mathbb{N}$ soient dans l'ensemble $\mathcal{L}$.*

Démonstration. La preuve est similaire à celle de la proposition 3.7 : on considère une exécution quelconque $x_0 \to x_1 \to ... \to x_n \to ...$, d'après la convergence il existe, par définition, un indice i_1 tel que $x_{i_1} \in \mathcal{L}$, on tronque ensuite le début de l'exécution et on itère le raisonnement. Ainsi, on montre qu'il existe une infinité d'indices $i_k \in \mathbb{N}$ tels que les x_{i_k} soient dans $\mathcal{L}$. □

3.6.4 Auto-stabilisation

L'auto-stabilisation est un concept introduit par Dijkstra en 1973 (voir les articles [Dij73, Dij74]), qui caractérise des systèmes capables par eux-mêmes de retrouver, après une panne ponctuelle, un comportement correct. Pour une présentation en détail du sujet, de ses enjeux et des problèmes que l'auto-stabilisation permet de résoudre, on peut se référer au livre de Dolev [Dol00].

L'auto-stabilisation est un cas particulier de la convergence. En effet, elle assure que, quel que soit l'état de départ, après une certaine phase dite de stabilisation, le système va arriver dans ensemble de "bons" états, appelé ensemble *légitime* (voir définition 3.14) et à partir de ce moment là, le système va avoir un comportement "correct" (conforme à une certaine spécification).

Définition 3.14. *[Tel94] Soit P une spécification sur les exécutions. On dit qu'un système de transitions $\mathcal{S}$ est* auto-stabilisant *vers la spécification P s'il existe un ensemble $\mathcal{L}$ de configurations dites* légitimes *tel que :*
- *d'une part le système $\mathcal{S}$ converge vers $\mathcal{L}$ (***convergence***)et,*
- *d'autre part, toute exécution partant d'une configuration de $\mathcal{L}$ satisfait P (***correction***).*

Comme on le voit dans la définition, un système n'est pas auto-stabilisant en soi, mais pour une spécification particulière. Cependant, dans la suite, quand il n'y aura pas d'ambiguïté sur la spécification, on pourra l'omettre.

Dans toute la suite de cette thèse, nous allons étudier des algorithmes pour lesquels la spécification P est du genre “on reste toujours dans l'ensemble de configurations E”. De plus, pour tous les algorithmes considérés l'ensemble E choisi va être fermé, à savoir :

Définition 3.15. *Étant donné un ensemble E et une relation de transition $\to$, on dit que E est* fermé [2] *pour $\to$ si, pour tout élément e de E on a : $e \to e'$ implique $e' \in E$*

Dans ce cas, pour montrer l'auto-stabilisation du système $\mathcal{S}$ vers P, on va montrer la convergence de $\mathcal{S}$ vers E, et bien sûr vérifier la fermeture de E pour le système $\mathcal{S}$. Ainsi on va montrer que l'ensemble de configurations $\mathcal{L} = E$ est légitime. Par abus de langage, nous allons également dire que le système $\mathcal{S}$ est auto-stabilisant vers $\mathcal{L} = E$, et non plus vers la spécification P qui qualifie l'ensemble des exécutions qui restent dans E.

Dans le cas où l'on n'impose pas que tout mot hors de $\mathcal{L}$ soit réductible, on va définir la notion suivante :

Définition 3.16. *On dit qu'un ensemble de configurations $\mathcal{L}$ est* absorbant *pour un système de transitions $\mathcal{S}$ si, d'une part, l'ensemble $\mathcal{L}$ est inévitable pour $\mathcal{S}$, d'autre part, l'ensemble $\mathcal{L}$ est fermé (pour $\mathcal{S}$).*

Cette notion d'états absorbants va nous servir dans le chapitre 8 à décomposer la preuve d'auto-stabilisation. Ainsi on peut montrer d'une part qu'un ensemble $\mathcal{L}$ est absorbant, et d'autre part que toute configuration hors de $\mathcal{L}$ est réductible. On obtiendra ainsi l'auto-stabilisation vers $\mathcal{L}$.

Exemple 3.17. La figure 3.4 présente un système de transitions très simple à 5 configurations. Il illustre la différence entre convergence et auto-stabilisation. Ainsi ce système converge

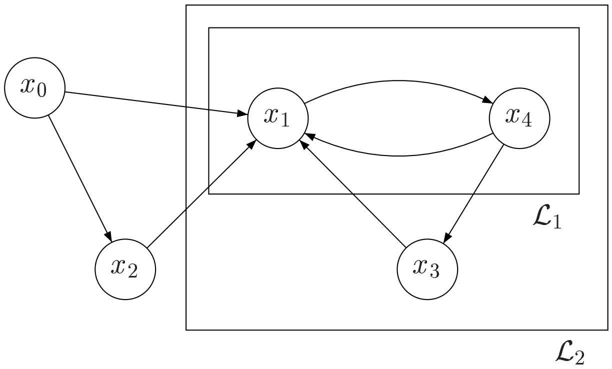

FIG. 3.4 – Un système de transitions convergeant vers $\mathcal{L}_1$ et auto-stabilisant vers $\mathcal{L}_2$.

vers l'ensemble $\mathcal{L}_1 = \{x_1, x_4\}$, car toute exécution infinie, quel que soit son état de départ, va passer par l'ensemble $\mathcal{L}_1$. Par contre, on n'a pas auto-stabilisation vers $\mathcal{L}_1$.
Raisonnons par l'absurde. Soit $\mathcal{L}$ un ensemble de configurations légitime selon la définition 3.14. D'après la propriété de correction, il ne peut pas contenir x_0 car il existe une exécution infinie issue de x_0 qui est infiniment souvent hors de $\mathcal{L}_1$: $x_0, x_2, x_1, x_4, x_3, x_1, x_4, x_3, \ldots$

[2] Dans la littérature, on trouve également le terme *clos* pour désigner la même notion. Le choix de *fermé* est délibéré, pour ne pas confondre avec la notion de *mots clos* qui sera utilisée au chapitre 7.

qui itère le cycle x_1, x_4, x_3. De même, en prenant la même exécution et en enlevant le premier état, on montre que x_2 n'est pas dans $\mathcal{L}$, puis en itérant qu'aucune configuration n'est dans $\mathcal{L}$. L'ensemble $\mathcal{L}$ est donc vide et ne peut pas vérifier la propriété de convergence. On aboutit donc à une contradiction, et on en déduit que le système n'est pas auto-stabilisant vers $\mathcal{L}_1$.

Par contre, du fait que l'ensemble $\mathcal{L}_2$ contient $\mathcal{L}_1$ et qu'il est fermé, on a auto-stabilisation du système de transitions vers l'ensemble $\mathcal{L}_2$ (en particulier l'ensemble $\mathcal{L}_2$ est lui-même légitime). ♣

Intérêt de l'auto-stabilisation

L'auto-stabilisation est une propriété qui est recherchée dans le cadre de la tolérance aux pannes. En effet, si l'on suppose qu'à un moment donné, suite à une panne transitoire, le système se trouve dans une configuration illégitime, on sait que, en l'absence de nouvelle panne, il va retrouver un comportement correct en un temps fini, puis le conserver.

On parle d'*auto*-stabilisation car ce retour à un comportement normal se fait sans aucune aide extérieure, que ce soit pour détecter la panne ou pour y remédier.

Il faut tout de même préciser que cette auto-stabilisation ne peut avoir lieu que tant que le code du programme et le comportement des différents processus ne sera pas affecté après la panne. On suppose donc que cette panne n'a fait, en fin de compte, que changer les états des processus qui, une fois la panne finie, se remettront à exécuter normalement le code.

Un exemple assez classique dans ce domaine est celui de l'exclusion mutuelle (voir section 3.1.4). Un algorithme auto-stabilisant d'exclusion mutuelle va assurer deux choses :
- si l'on part d'une configuration quelconque, on va arriver en un temps fini dans une configuration à un seul "jeton", et
- quand on a atteint ce genre de configuration, alors le système va vérifier les propriétés d'exclusion mutuelle (*cf.* section 3.1.4).

Certains algorithmes d'exclusion mutuelle seront présentés dans cette thèse, comme par exemple celui de Herman [Her90], présenté à l'exemple 3.20, page 62, et prouvé suivant notre méthode à l'exemple 4.3, page 69, celui de Kakugawa et Yamashita [KY97], présenté dans la partie 4.2.3 fonctionnant sous un ordonnancement centralisé quelconque, ou encore celui de Beauquier, Gradinariu et Johnen [BGJ99a], présenté dans la partie 4.2.2 et fonctionnant également sous un ordonnancement centralisé quelconque.

Pour plus d'informations sur l'auto-stabilisation en général, ou l'exclusion mutuelle auto-stabilisante en particulier, on peut se rapporter en français à la thèse de Sylvie Delaët [Del95]. En anglais, on peut citer l'article de synthèse de Schneider [Sch93] ou le livre de Dolev [Dol00].

3.7 Systèmes paramétrés et indécidabilité

Lorsque l'on s'intéresse à un système distribué fixé et fini, on peut décider des propriétés comme l'accessibilité ou la convergence, du fait que le nombre d'états, et par conséquent le système de transitions sous-jacent, est fini. Pour cela, on peut parfois faire appel à des outils automatiques de vérification, appelés aussi model-checkers. Selon la façon dont les différentes entités du système communiquent entre elles, on peut représenter le système sous forme d'un réseau d'automates, ou d'automates communicants lorsque l'on considère des échanges de messages asynchrones. Ainsi on peut utiliser l'outil approprié, comme par exemple SMV pour des automates dont la communication s'effectue par variables partagées, ou bien SPIN pour le cas

des automates communicants.
Une présentation entre autres de ces deux outils de model-checking, ainsi que des références pour chaque outil peuvent être trouvées dans le livre [BBF$^+$01], ou sa version française [SBB$^+$99]. Dans le cadre probabiliste, on peut citer le model-checker PRISM [KNP02], qui permet de vérifier des propriétés de pCTL sur des systèmes distribués finis[3].

Le problème se pose lorsque, comme on va le considérer dans la suite, on veut montrer qu'une propriété est vraie, et ce quel que soit le nombre N de processus dans le système. On va donc considérer des systèmes finis, dont le nombre de processus est constant le long d'une exécution, mais pour lesquels ce nombre n'est pas connu à l'avance.

Cette notion de paramètre fini mais non borné introduit une source d'infini. En effet, on doit alors vérifier un nombre infini dénombrable de systèmes finis. On n'a donc plus de résultat de décidabilité dans le cas général. Ainsi, vérifier qu'un système distribué converge vers un certain ensemble d'états, et ce quel que soit le nombre de processus dans le système, est indécidable (voir l'article [AK86]).

Face à ce problème, nous avons eu deux approches différentes. La première a été de faire une preuve en laissant une part non mécanisée, la construction d'une mesure qui décroît à chaque étape, comme on le verra dans la partie II pour des systèmes probabilistes. La deuxième consiste à trouver un algorithme qui permet de montrer automatiquement la propriété voulue sur une sous-classe de ces systèmes paramétrés. Cette deuxième approche est celle considérée dans la partie III, sur des systèmes non probabilistes.

D'autres travaux ont été faits sur la vérification de systèmes distribués paramétrés. Parmi ceux-ci, on peut citer la méthode des invariants de réseau, développée dans [KM95], qui a été utilisée par Pnueli, Zuck *et al* [KPSZ02, ZPK02, APZ03] pour prouver la convergence de l'algorithme du dîner des philosophes courtois (introduit dans [LR81]) en mêlant invariants de réseau et model-checker probabiliste.
La méthode d'accélération est également beaucoup utilisée dans ce cadre, principalement pour prouver des propriétés de sûreté ([JN00, BJNT00, Tou01]), mais également ponctuellement des propriétés de vivacité ([PS00]) en recherchant des éventuels "mauvais" cycles dans les exécutions.
Certains travaux donnent également un critère décidable permettant de vérifier automatiquement les propriétés voulues. L'article [BMT01] montre que si d'une part les règles de réécriture sont des permutations, et si d'autre part l'ensemble de départ est d'une forme particulière – les Alphabetic Pattern Constraints– alors on peut calculer automatiquement l'image de cet ensemble par application itérée des règles. L'article [BJNT00] donne pour sa part un critère permettant de calculer directement la relation de transition itérée, afin de vérifier des propriétés de sûreté ou de vivacité. Enfin, l'article [BBFM01], auquel est consacré le chapitre 7, présente une méthode de preuve de convergence utilisant des techniques de réécriture, et plus particulièrement les surréductions.

Dans ce contexte des systèmes paramétrés, on a aussi besoin de définir les ensembles de configurations de manière paramétrée. En effet, si l'on reprend l'exemple des propriétés de convergence sur un système en anneau, l'ensemble vers lequel on veut que le système converge est soit infini, soit la convergence n'est pas vérifiée. Si cet ensemble était fini, il ne contiendrait pas des "mots" de toutes les longueurs, et donc il existerait une taille d'anneau à partir de

[3] Le programme, ainsi que de nombreux cas d'étude sont disponibles en ligne, sur la page `http://www.cs.bham.ac.uk/~dxp/prism/`

laquelle le système ne convergerait plus. Ainsi dans la suite on va souvent représenter nos ensembles de configurations sous forme de langages réguliers.

Exemple 3.18. Ainsi, le langage des configurations ayant au moins un processus dans l'état a sera noté $\Sigma^* a \Sigma^*$, et si $\Sigma = \{0, 1\}$, l'ensemble des configurations dans lesquelles deux processus consécutifs sont dans un état différent est noté $\{1 \cup \varepsilon\}.(01)^*.\{0 \cup \varepsilon\}$. ♣

3.8 Systèmes distribués probabilistes

Outre le cas des systèmes intrinsèquement probabilistes, les probabilités ont été introduites dans les systèmes distribués pour principalement deux raisons. D'un côté elles permettent de résoudre plus efficacement certains problèmes, et d'un autre côté elles permettent de résoudre d'autres problèmes dont on ne connaissait pas de solution dans le cadre déterministe, voire dont on avait prouvé qu'il n'existait pas de solution déterministe. Un exemple de ce dernier cas est l'algorithme du dîner des philosophes, pour lequel il a été prouvé qu'il n'existait pas de solution déterministe, symétrique et totalement distribuée (voir [RL94] et le chapitre 5).

Ainsi il a fallu adapter le cadre des systèmes distribués à cet ajout de probabilités.
Lorsque l'on introduit des probabilités dans un système distribué, on retrouve le modèle de processus de décision markovien décrit à la section 2.3. Dans ce cas, c'est le démon qui est source de non-déterminisme.
Les actions du processus de décision markovien correspondent ici aux choix non déterministes du démon entre les (sous-ensembles de) positions activables. Les distributions de probabilités que l'on avait dans le processus de décision markovien correspondent ici aux choix probabilistes entre les différents résultats possibles des règles probabilistes.

Exemple 3.19. Considérons un système distribué probabiliste à N processus disposés en anneau, dont l'espace d'états est $\Sigma = \{0, 1, 2\}$, et défini par les deux règles probabilistes suivantes :

$$
\begin{array}{lcl}
01 & \to & \left\{ \begin{array}{l} 11 \text{ avec probabilité } 1/2 \\ 21 \text{ avec probabilité } 1/2 \end{array} \right. \\
02 & \to & \left\{ \begin{array}{l} 12 \text{ avec probabilité } 1/3 \\ 22 \text{ avec probabilité } 2/3 \end{array} \right.
\end{array}
$$

Comme les règles changent la première des deux lettres concernées, c'est donc à cette position que l'on va considérer que la règle est appliquée. Supposons que le système possède un démon distribué. Partant de la configuration $x = 01020$, il y a deux processus activables, en positions 0 et 2. Si le démon sélectionne ces deux positions, alors la distribution de probabilité sur l'ensemble des configurations suivantes possibles est :

$$
\begin{array}{ll}
11120 & \text{avec probabilité } 1/6 \\
11220 & \text{avec probabilité } 1/3 \\
21120 & \text{avec probabilité } 1/6 \\
21220 & \text{avec probabilité } 1/3
\end{array}
$$

♣

Le problème pour définir un espace de probabilités sur les exécutions réside, comme on l'a vu dans la section 2.3, dans les choix non-déterministes du démon. C'est pourquoi on va appliquer ici la même méthode que celle présentée dans la section 2.3, à savoir fixer les choix non-déterministes au moyen d'un ordonnancement pour pouvoir définir un espace de probabilités sur les exécutions.

Le fait d'introduire la notion d'ordonnancement fait passer tous les choix non déterministes du démon au début de l'exécution : ainsi le démon ne choisit plus à chaque étape les positions à réécrire de manière non déterministe, mais choisit une fois pour toutes un ordonnancement parmi ceux qui lui sont possibles.

Une fois que l'on a fixé l'ordonnancement, on a un arbre d'exécutions probabilistes, sur lequel on va pouvoir définir notre espace de probabilités.

Lorsque l'on voudra vérifier qu'une propriété est satisfaite sur le système quels que soient les choix du démon, il suffira de la montrer quel que soit l'ordonnancement.

Les différentes caractérisations décrites pour les démons –centralisé, distribué et synchrone– s'appliquent de manière naturelle aux ordonnancements. On peut remarquer qu'un démon synchrone ne possède qu'un seul ordonnancement, qui consiste à choisir à chaque étape tous les processus activables.

Un exemple de tel ordonnancement synchrone est donné dans l'exemple suivant, qui présente un algorithme auto-stabilisant[4] d'exclusion mutuelle créé par Herman, fonctionnant dans le cas où la taille N de l'anneau est impaire.

Exemple 3.20. Dans l'algorithme d'exclusion mutuelle d'Herman [Her90], le démon est synchrone et il n'existe donc qu'un ordonnancement $\mathcal{O}$. L'ensemble d'états est $\Sigma = \{0, 1\}$, et le nombre de processus N est impair.
Le système de transitions $\rightarrow$ est défini par les quatre règles suivantes :

$$\begin{array}{lcl} 01 & \rightarrow & 00 \\ 10 & \rightarrow & 11 \\ 11 & \rightarrow & \begin{cases} 11 \text{ avec probabilité } 1/2 \\ 10 \text{ avec probabilité } 1/2 \end{cases} \\ 00 & \rightarrow & \begin{cases} 00 \text{ avec probabilité } 1/2 \\ 01 \text{ avec probabilité } 1/2 \end{cases} \end{array}$$

Contrairement à l'exemple 3.19, ici c'est la deuxième lettre qui peut être modifiée. C'est donc la position de cette deuxième lettre qui compte lorsqu'on décrit où l'on applique la règle.
On peut tout d'abord remarquer que, même si les règles déterministes consistent à copier l'état du voisin de gauche, elles n'ont pas pour effet d' "uniformiser" les états des processus. Étant donné que, lors de chaque transition, on applique une règle à chaque position, si on a dans une configuration x une suite de la forme 01010101, elle va se transformer en a0101010 où a dépend de la lettre qui précédait le premier 0 de ce motif dans la configuration. L'application des règles non probabilistes n'uniformise donc pas l'état des différents processus, mais préserve plutôt ces suites alternées.

Pour chaque configuration, toutes les positions sont activables : lorsqu'un processus est dans le même état que son voisin de gauche, on peut lui appliquer une règle probabiliste, et s'ils sont dans des états différents, on peut lui appliquer une règle déterministe.

Si l'on considère par exemple la configuration $x = 00101$, comme l'ordonnancement va sélectionner tous les processus, on va appliquer une règle probabiliste (qui va modifier (ou non) la lettre à la position 1) et quatre règles déterministes (partout ailleurs). On a donc deux

[4]Pour une définition d'auto-stabilisation, voir la section 3.6.4.

possibilités de transition : $x \xrightarrow{\mathcal{O}} x_1 = 11010$ et $x \xrightarrow{\mathcal{O}} x_2 = 10010$, chacune de probabilité 1/2.
♣

La partie II présente une méthode de vérification de propriétés de convergence sur des systèmes distribués, illustrée sur plusieurs exemples dont l'algorithme d'Herman décrit ci-dessus, qui sera prouvé dans l'exemple 4.3, page 69.

Le chapitre 5 va s'intéresser plus précisément à un algorithme : une variante sans équité du dîner des philosophes probabilistes, et prouver sur celui-ci une propriété de convergence probabiliste que l'on appellera *progrès* (voir section 5.5).

3.8.1 Logique temporelle probabiliste

Même si nous n'exprimons pas dans la suite nos propriétés à vérifier à l'aide d'une logique formelle, nous présentons ici la logique pCTL (Probabilistic Computational Tree Logic), car elle permet d'exprimer un résultat permettant de restreindre l'ensemble des ordonnancements à considérer aux ordonnancements déterministes ou aux ordonnancements sans mémoire. Cette section vise donc à introduire pCTL afin d'amener ce résultat.

Pour faire de la vérification sur des systèmes probabilistes et exprimer formellement les propriétés que l'on veut vérifier, on a besoin d'une logique appropriée. Il faut tout d'abord une logique temporelle pour modéliser des propriétés sur des exécutions du genre :

Si on a φ_1 alors il existe un chemin le long duquel φ_2 est toujours vraie.

Comme on travaille sur des systèmes probabilistes, on va adjoindre à cette logique un opérateur probabiliste.

La logique pCTL est une extension de CTL (voir [Eme90]), à laquelle on a ajouté un opérateur probabiliste $\mathbb{P}$, s'utilisant sous la forme $\mathbb{P}_{\leq a}\varphi$ (resp. $\mathbb{P}_{\geq a}\varphi$) et signifiant que la probabilité que la formule φ soit vraie au cours des évolutions futures du système est inférieure (resp. supérieure) à une certaine valeur a.

Dans cette logique, comme dans son homologue non probabiliste CTL, on distingue deux classes de formules : la classe des formules d'états, $Stat$, dont la véracité est évaluée sur les états, et la classe des formules de chemins, Seq, dont la véracité peut être évaluée sur les chemins (infinis).

Remarque : Comme nous considérons des systèmes distribués, les "états" décrits ci-dessus correspondent aux configurations du système distribué. De même, les "chemins" que nous allons considérer sont en fait des exécutions du système.

Syntaxe

La syntaxe de la logique pCTL est la suivante :

$$\begin{array}{rcl} \mathcal{P} & \subseteq & Stat \\ \varphi, \psi \in Stat & \Rightarrow & \varphi \wedge \psi, \neg\varphi \in Stat \\ \varphi \in Seq & \Rightarrow & A\varphi, E\varphi, \mathbb{P}_{\sim a}\varphi \in Stat \\ \varphi, \psi \in Stat & \Rightarrow & X\varphi, G\varphi, F\varphi, \varphi \mathcal{U} \psi \in Seq \end{array}$$

Dans la définition ci-dessus, $\mathcal{P}$ représente l'ensemble des propositions atomiques, $\sim$ désigne un opérateur de comparaison appartenant à l'ensemble $\{<, >, \leq, \geq\}$, et a désigne une probabilité dans l'intervalle $[0, 1]$.

Sémantique

Nous allons à présent donner la sémantique des différents opérateurs décrits ci-dessus. Étant donnée une formule de chemins $\varphi \in Seq$, la notation $\omega \models \varphi$ signifie que le chemin ω satisfait φ. Pour une formule d'états $\varphi \in Stat$, la notation $s \models \varphi$ signifie que l'état s satisfait la formule φ.

La sémantique des opérateurs booléens ainsi que des opérateurs temporels X (dans le prochain état) G (toujours), F (un jour) et $\mathcal{U}$ (jusqu'à ce que) est définie de manière usuelle (voir [Eme90]).

La sémantique des autres opérateurs de pCTL est la suivante :

$$\begin{array}{rcl} s \models A\varphi & \Leftrightarrow & \forall \omega \in \Omega_s, \omega \models \varphi \\ s \models E\varphi & \Leftrightarrow & \exists \omega \in \Omega_s : \ \omega \models \varphi \\ s \models \mathbb{P}_{\leq a}\varphi & \Leftrightarrow & \mathbb{P}^+_s(\{\omega \in \Omega_s | \omega \models \varphi\}) \leq a \\ s \models \mathbb{P}_{\geq a}\varphi & \Leftrightarrow & \mathbb{P}^-_s(\{\omega \in \Omega_s | \omega \models \varphi\}) \geq a \end{array}$$

On définit de manière similaire la sémantique de $\mathbb{P}_{<a}$ et $\mathbb{P}_{>a}$.

Étant donnés un PDM et une formule logique probabiliste, donnée en pCTL, on peut vouloir rechercher des ordonnancements qui minimisent ou maximisent la probabilité de satisfaire la formule. Formellement :

Définition 3.21. *Soient $s \in S$ un état d'un PDM, et ϕ une propriété sur les exécutions. Un ordonnancement $\mathcal{O}$ est dit* le plus favorable *(resp.*le plus défavorable*) pour ϕ si :* $\mathbb{P}_{s,\mathcal{O}}(\{\omega \in \Omega_s | \omega \models \phi\}) = \mathbb{P}^+_s(\phi)$ *(resp.* $\mathbb{P}_{s,\mathcal{O}}(\{\omega \in \Omega_s | \omega \models \phi\}) = \mathbb{P}^-_s(\phi)$ *)*

Le théorème suivant, tiré de [BdA95], montre que l'on peut se restreindre aux ordonnancements particuliers de la définition 2.25.

Théorème 3.22. *Pour tout PDM et toute formule de chemins ϕ de pCTL, il existe des ordonnancements déterministes et des ordonnancements sans mémoire qui sont les plus favorables et les plus défavorables pour tout état $s \in S$.*

Étant donnée une formule de pCTL, on va donc pouvoir calculer pour tout état s les probabilités $\mathbb{P}^+_s(\{\omega \in \Omega_s | \omega \models \phi\})$ et $\mathbb{P}^-_s(\{\omega \in \Omega_s | \omega \models \phi\})$ en considérant uniquement les ordonnancements déterministes.

Dans la partie II où l'on va s'intéresser à des systèmes distribués probabilistes, nous allons utiliser le théorème 3.22 pour ne considérer que des ordonnancements déterministes.

3.8.2 Convergence probabiliste

La convergence probabiliste a déjà été définie dans le chapitre 2, mais nous allons donner ici la notation que nous allons utiliser dans la partie II sur des systèmes distribués probabilistes.

Étant donnés un système $\mathcal{S}$, un ordonnancement $\mathcal{O}$ et un ensemble de configurations $\mathcal{L} \subseteq X$, la propriété de convergence probabiliste de $\mathcal{S}$ vers $\mathcal{L}$, habituellement notée

$$\forall x \in X, \ \mathbb{P}_{x,\mathcal{O}}(\sigma = x_0, J_0, ..., x_n, J_n, ... \in \Omega_{x,\mathcal{O}} | \exists i : \ x_i \in \mathcal{L}) = 1$$

(où chaque J_i est un ensemble de positions activables dans la configuration x_i choisi par $\mathcal{O}$) sera abrégée en :

$$\forall x \in X, \ \ \mathbb{P}(x \underset{\mathcal{S}}{\overset{\mathcal{O}}{\longrightarrow}}{}^* \mathcal{L}) = 1.$$

Deuxième partie

Convergence de systèmes paramétrés probabilistes

Chapitre 4

Critère formel de convergence

Dans ce chapitre, dont le contenu est principalement tiré de [DFP01], nous allons présenter, et illustrer sur des exemples, notre méthode de preuve de convergence sur des systèmes paramétrés probabilistes. Celle-ci se base sur le fait que, pour prouver la convergence probabiliste d'un système, il n'est pas nécessaire de montrer que, à partir de toute configuration on arrive en un temps fini avec probabilité 1 dans l'ensemble voulu, mais il suffit de montrer qu'on atteint cet ensemble avec probabilité non nulle en un temps fini, à partir de toute configuration.

Nous exposons d'abord à la section 4.1 cette méthode de preuve dans le cas où l'ordonnancement est fixé, déterministe et sans mémoire et où par conséquent le système se comporte comme une chaîne de Markov. Nous expliquons ensuite comment ce résultat peut s'étendre au cas d'un ordonnancement arbitraire à la section 4.2. La section 4.3 présente une façon de simplifier la preuve, qui consiste à trouver une fonction φ qui ne croît jamais le long d'une exécution, puis de raisonner à φ constante. Enfin nous montrons comment, en utilisant la notion de *lumping* des chaînes de Markov, on peut restreindre l'ensemble d'états à considérer, et donc calculer plus efficacement un temps moyen de convergence, à la section 4.4.

4.1 Algorithmes probabilistes vus comme des chaînes de Markov

Nous supposons dans cette section que l'ordonnancement $\mathcal{O}$ est donné, déterministe et sans mémoire. Ainsi pour toute exécution finie $x_0 \xrightarrow{J_0} x_1 \xrightarrow{J_1} \dots \xrightarrow{J_{\ell-1}} x_\ell$, $\mathcal{O}$ choisit de manière déterministe un sous-ensemble J_ℓ de processus activables de x_ℓ, qui ne dépend que de x_ℓ et pas des transitions ou des configurations précédentes.

Remarque : Dans toute cette partie, nous allons considérer des systèmes qui seront de deux types : soit les règles ne modifient qu'une position à la fois, et donc l'application de plusieurs règles simultanément à des positions différentes ne va pas générer de conflit, soit on ne considère que des ordonnancements centralisés, auquel cas pas de conflit non plus car une seul règle appliquée à la fois. Ainsi, dans chacun des deux cas il n'y aura pas de conflit, comme annoncé à la section 3.5.2.

Comme nous allons considérer des systèmes sans blocage, J_ℓ est non vide et la modification des lettres de position incluse dans J_ℓ change aléatoirement x_ℓ en l'une des configurations possibles $x_{\ell+1}^1, \cdots, x_{\ell+1}^n$ avec les probabilités associées $p(x_\ell \xrightarrow{J_\ell} x_{\ell+1}^1), \cdots, p(x_\ell \xrightarrow{J_\ell} x_{\ell+1}^n)$ de somme 1. Plus précisément, étant donnée une configuration $x \in X$, considérons l'ensemble I_x

de tous les couples (y, p) de $X \times [0, 1]$ tels que x se réécrit en y suivant $\mathcal{O}$ avec probabilité p. On a pour tout $x \in X$, $\Sigma_{(y,p)\in I_x}\ p = 1$. Le système se comporte donc comme une chaîne de Markov (pour les définitions associées, voir section 2.2).

De ce fait, on peut appliquer un résultat classique de la théorie de Markov (voir théorème 2.21) : quelle que soit la configuration initiale, la probabilité d'atteindre une configuration récurrente (*i.e.* $\in \mathcal{R}ec$) en un nombre fini de transitions est 1. Dans le contexte des systèmes distribués, ce théorème peut être reformulé de la façon suivante :

Théorème 4.1. *Étant donné un ordonnancement $\mathcal{O}$, et un système $\mathcal{S}$ se comportant suivant $\mathcal{O}$ comme une chaîne de Markov, on a :*
pour toute configuration x, $\mathbb{P}(x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^{*}\mathcal{R}ec) = 1$.

On va utiliser ce résultat pour prouver la convergence de certains algorithmes distribués vers un ensemble $\mathcal{L}$. En particulier, si l'ensemble $\mathcal{L}$ des configurations légitimes contient $\mathcal{R}ec$, on a convergence probabiliste vers $\mathcal{L}$.

Nous allons à présent donner une condition locale appelée Prop, qui assure que toute configuration non légitime est transitoire. Cette condition est locale car elle n'implique que des réductions en une étape $x \xrightarrow[\mathcal{S}]{\mathcal{O}} y$ (et non $x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^{*} y$).

Pour chaque valeur de N, supposons que l'on a une fonction D définie sur l'ensemble des configurations $X = \Sigma^N$ et une relation d'ordre $\ll$ sur $D(X)$. La condition Prop est définie comme suit :

Prop : $\forall x \notin \mathcal{L}\ \exists y :\ (x \xrightarrow[\mathcal{S}]{\mathcal{O}} y \wedge (y \in \mathcal{L} \vee D(y) \ll D(x)))$.

Cette condition signifie que pour toute configuration x non légitime, il existe une transition suivant $\mathcal{O}$ de x vers une configuration y qui est soit légitime soit plus petite pour D.

On a :

Théorème 4.2. *Étant donné un système $\mathcal{S}$ et un ordonnancement $\mathcal{O}$, s'il existe une fonction D et un ordre $\ll$ tels que*

Prop : $\forall x \notin \mathcal{L}\ \exists y :\ (x \xrightarrow[\mathcal{S}]{\mathcal{O}} y \wedge (y \in \mathcal{L} \vee D(y) \ll D(x)))$

alors $\forall x\ \ \mathbb{P}(x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^{*}\mathcal{L}) = 1$.

La preuve de ce résultat n'est pas donnée ici, car elle se fait en utilisant la même idée que la preuve du théorème 4.4, qui peut être vu comme une généralisation de ce théorème.

Remarque : Il est à noter que la condition Prop implique que le système n'a pas de blocage hors de $\mathcal{L}$. En effet, Prop affirme entre autres que toute configuration x n'appartenant pas à $\mathcal{L}$ possède un successeur y pour l'ordonnancement $\mathcal{O}$.

Étant donné un ordonnancement $\mathcal{O}$ déterministe et sans mémoire, la partie cruciale de la preuve de convergence réside dans le fait de trouver une fonction D et un ordre $\ll$ qui satisfont Prop.

Stratégie de réécriture. Dans Prop, la quantification existentielle sur y ne représente pas un choix des positions réécrites, car ce choix est déterminé par $\mathcal{O}$. Elle ne correspond pas non plus au choix d'une règle, car dans les systèmes que nous allons considérer, une fois que les positions à réécrire sont choisies, il n'y a pas d'ambiguïté sur la règle à appliquer. Cette quantification porte uniquement sur le résultat de l'application des règles probabilistes. Ce que

l'on s'autorise, pour montrer qu'il existe un chemin suivant $\mathcal{O}$ qui décroît pour D à chaque étape, c'est de "choisir" le résultat des tirages aléatoires. Ce que l'on va faire c'est exhiber une stratégie de réécriture des règles probabilistes, que l'on appellera pour simplifier *stratégie de réécriture*, qui donne pour chaque configuration x un successeur possible y suivant $\mathcal{O}$ tel que $D(y) \ll D(x)$.
Cette stratégie de réécriture prend parfois en compte des informations comme les indices des processus, ou l'état de processus assez éloigné ; informations auxquelles les processus eux-mêmes n'ont souvent pas accès.

Exemple 4.3. Nous allons maintenant appliquer le théorème 4.2 pour prouver que l'algorithme de Herman décrit dans l'exemple 3.20 converge. Une preuve de ce résultat est donnée dans l'article original [Her90], mais demande plus d'efforts pour prouver directement la convergence avec probabilité 1. Ici on va juste exhiber une stratégie de réécriture, et montrer qu'elle fait décroître D.

La notation $\bar{q}$ signifie $q+1$ où $+$ est l'addition modulo 2. Rappelons que les opérations sur les indices sont faites modulo N. Pour tout mot $u = q_0q_2...q_{N-1} \in \Sigma^N$, la notation $\bar{u}$ désigne le mot $\bar{q}_0\bar{q}_1...\bar{q}_{N-1}$.

Étant donnée une configuration $x = q_0q_1 \cdots q_{N-1}$, on dit qu'il y a un "jeton" à la position i dans x si $q_i = q_{i-1}$. La fonction D que l'on va choisir possède deux composantes : $D_1 = card(\{i \in \{0, \cdots, N-1\} \mid q_i = q_{i-1}\})$ qui compte le nombre de jetons dans une configuration, et D_2 qui vaut la distance minimale entre deux jetons consécutifs (N lorsque la configuration compte strictement moins de deux jetons). On peut remarquer que, étant donné que l'anneau possède un nombre impair de processus, il y a toujours au moins un jeton dans une configuration.

L'ensemble légitime se compose ici de toutes les configurations ayant exactement un jeton ($D_1 = 1$). On cherche donc à montrer que cet algorithme converge vers un ensemble de configurations vérifiant la propriété d'exclusion mutuelle : à terme seul un processus possédera un jeton, correspondant à l'accès exclusif à une ressource.

L'ordre $\ll$ est ici l'extension lexicographique de $<$ à savoir que $D(y) \ll D(x) \Leftrightarrow D_1(y) < D_1(x) \vee (D_1(y) = D_1(x) \wedge D_2(y) < D_2(x))$.

Notre stratégie de réécriture est la suivante : pour chaque configuration on dénote par k la position du jeton tel que la distance entre lui-même et le plus proche jeton à sa droite est la plus petite (lorsqu'il y en a plusieurs, on prend le plus petit indice k possible). Ensuite, lors de la transition, on réécrit tous les jetons en changeant leur état, sauf celui en position k dont l'état reste inchangé. Ainsi, comme on inverse l'état de tous les processus (sauf k) tous les jetons restent en place (sauf celui en k) : si deux processus consécutifs étaient dans la même état aa ils passent à $\bar{a}\bar{a}$ et le jeton est conservé. De même s'il n'y avait pas de jeton on n'en crée pas. Pour ce qui est du jeton en position k, il se décale d'une place vers la droite.

Par exemple si $x = 00110101101$, x possède trois jetons, en positions 1, 3 et 8. La plus petite distance est entre 1 et 3. Notre stratégie consiste donc à réécrire x en $y = 10001010010$. On a toujours trois jetons ($D_1(x) = D_1(y)$), mais la plus petite distance est à présent 1, entre les deux jetons en position 2 et 3 ($D_2(y) < D_2(x)$).

Lorsque deux jetons sont contigus c'est-à-dire lorsque leur distance est 1 (comme dans y), l'application de notre stratégie va "fusionner" les deux jetons, ce qui va les faire disparaître tous les deux. Si l'on applique cela à y on obtient $z = 01010101101$ qui ne possède plus qu'un seul jeton en position 8 et qui appartient à $\mathcal{L}$.

Ainsi on a bien une stratégie de réécriture qui va faire décroître D à chaque étape tant que l'on n'arrive pas dans $\mathcal{L}$. D'après le théorème 4.2, on a donc convergence de cet algorithme vers l'ensemble $\mathcal{L}$ des configurations satisfaisant l'exclusion mutuelle. ♣

D'autres algorithmes peuvent être prouvés de la même manière, comme ceux de Beauquier et Delaët [BD94] ou Flatebo et Datta [FD94].

4.2 Cas d'un ordonnancement arbitraire

Considérons à présent le cas où le mécanisme de choix (*i.e.* le démon) peut décider à chaque étape de manière arbitraire les processus activables sélectionnés. Supposons également qu'il possède une mémoire, potentiellement illimitée, et peut donc choisir le(s) prochain(s) processus sélectionné(s) en fonction de la totalité du comportement passé. Le système peut alors se décrire comme un processus de décision markovien (défini à la section 2.3), à savoir une alternance de choix non déterministes (les processus sélectionnés) et de transitions probabilistes (le résultat de l'application des règles probabilistes et déterministes).

Comme annoncé à la section 2.3, on ne peut pas définir de probabilités directement sur l'ensemble d'exécutions engendré par un tel système. Pour avoir un espace de probabilités, il faut fixer un ordonnancement et considérer l'ensemble des exécutions suivant cet ordonnancement. On va ensuite, dans les preuves de convergence, considérer tous les ordonnancements possibles.

Dans la section 3.8.1, nous avons vu que, d'après le théorème 3.22 dû à Bianco et de Alfaro, il est suffisant de considérer les ordonnancements déterministes, lorsque la propriété à vérifier est exprimable en pCTL. Comme la convergence probabiliste est typiquement exprimable en pCTL, nous allons nous restreindre aux ordonnancements déterministes.

La propriété Prop du théorème 4.2 (dépendante d'un ordonnancement particulier) doit ici être renforcée pour prendre en compte tous les ordonnancements possibles (voir Prop' ci-après). En suivant la formulation du théorème 4.2 on a, dans ce contexte d'ordonnancement arbitraire :

Théorème 4.4. *Étant donné un système de réécriture $\mathcal{S}$, supposons qu'il existe une fonction D et un ordre $\ll$ tels que*
Prop' : $\forall x \notin \mathcal{L} \quad \forall J \subseteq \mathcal{E}(x)\ \exists y : \ (x \xrightarrow[\mathcal{S}]{J} y \wedge (y \in \mathcal{L} \vee D(y) \ll D(x)))$
alors pour tout ordonnancement $\mathcal{O}$, $\forall x$ $\ \mathbb{P}(x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^{*} \mathcal{L}) = 1$.

Remarque : Lorsque l'on s'intéresse uniquement (par exemple pour éviter des conflits lorsque les actions modifient plus d'une lettre) à des ordonnancements centralisés, on a un théorème de convergence similaire, où le $\forall J \subseteq \mathcal{E}(x)$ est remplacé par $\forall i \in \mathcal{E}(x)$, et où la conclusion n'est vraie que pour les ordonnancements centralisés. La preuve est identique.

La preuve du théorème 4.4 nécessite quelques préliminaires :

On dit qu'une exécution ρ *traverse $\mathcal{L}$ en m étapes* si ρ est une exécution de la forme $x_0 \xrightarrow{J_0} \cdots \xrightarrow{J_{i-1}} x_i \xrightarrow{J_i} \cdots$ telle que $\exists j \in \{0, ..., m\} : \ x_j \in \mathcal{L}$.

Étant donnés un ordonnancement $\mathcal{O}$, une configuration initiale x_0 et un entier $m \geq 0$, l'ensemble des configurations de $\Omega_{x_0,\mathcal{O}}$ traversant $\mathcal{L}$ en m étapes est doté d'une probabilité : $\mathbb{P}_{x_0,\mathcal{O}}(\{\rho = x_0, J_0, x_1, J_1, ..., x_n, J_n, ... \in \Omega_{x_0,\mathcal{O}} \mid \rho \text{ traverse } \mathcal{L} \text{ en } m \text{ étapes }\})$, abrégée en

$\mathbb{P}(x_0 \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^{\leq m}\mathcal{L})$. Celle-ci est bien définie car l'ensemble de ces exécutions est une union finie de cylindres.

Montrons à présent que lorsque Prop' est satisfaite, $lim_{m\to\infty}\ \mathbb{P}(x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^{\leq m}\mathcal{L}) = 1$, et donc $\mathbb{P}(x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^{*}\mathcal{L}) = 1$.

Lemme 4.5. *Considérons un système $\mathcal{S}$ sans blocage, une fonction D et un ordre $\ll$ tels que* Prop' : $\forall x \notin \mathcal{L}\ \ \forall J \subseteq \mathcal{E}(x)\ \exists y :\ (x \xrightarrow[\mathcal{S}]{J} y \wedge (y \in \mathcal{L} \vee D(y) \ll D(x)))$, *alors il existe un entier $M > 0$ et une probabilité $p > 0$ telle que, pour tout ordonnancement $\mathcal{O}$,*

$$\forall x\ \ \mathbb{P}(x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^{\leq M}\mathcal{L}) \geq p.$$

Démonstration. Soit M le nombre d'éléments de X (i.e. $M = |X| = |\Sigma|^N$). Soit q la probabilité minimale associée à une règle probabiliste. Étant donné un ordonnancement $\mathcal{O}$ et une configuration initiale x_0, considérons une exécution ρ suivant $\mathcal{O}$ de la forme $x_0 \xrightarrow[\mathcal{S}]{} x_1 \xrightarrow[\mathcal{S}]{} \cdots$ telle que, pour tout $k \geq 0$, si ρ n'a pas traversé $\mathcal{L}$ en k étapes (i.e. $x_0 \notin \mathcal{L}, \cdots, x_k \notin \mathcal{L}$), alors $x_{k+1} \in \mathcal{L}$ ou $D(x_{k+1}) \ll D(x_k)$.

Notons que, étant donné que l'on suppose Prop' vraie, une telle exécution ρ existe. D'après le lemme des tiroirs, parmi les $M+1$ premières configurations $x_0, \cdots, x_M$ de ρ, il y a nécessairement deux éléments x_i et x_j qui sont identiques. Par conséquent ρ traverse $\mathcal{L}$ en M étapes, car sinon on aurait $D(x_i) \ll D(x_{i+1}) \ll \cdots \ll D(x_j)$, ce qui est impossible car $x_i = x_j$. D'autre part, chaque transition de ρ a une probabilité au moins égale à q. La probabilité de l'exécution finie composée des M premières transitions de ρ est donc au moins égale à q^M. On en déduit que la probabilité de l'ensemble des exécutions issues de x_0 suivant $\mathcal{O}$ qui traversent $\mathcal{L}$ en M étapes est au moins $p = q^M$. Ainsi pour tout $\mathcal{O}$ et tout x_0, $\mathbb{P}(x_0 \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^{\leq M}\mathcal{L}) \geq p$. □

Démonstration. (théorème 4.4) Considérons un ordonnancement $\mathcal{O}$, arbitraire mais fixé. Étant donné que l'on suppose que la relation de transition $\xrightarrow[\mathcal{S}]{\mathcal{O}}$ satisfait Prop', on a d'après le lemme 4.5 : $\forall x \in X,\ \mathbb{P}(x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^{\leq M}\mathcal{L}) \geq p$. Ainsi, quelle que soit la configuration x_0 dont on part, la probabilité de ne pas traverser $\mathcal{L}$ en M étapes est au plus $1-p$. En itérant ces arguments, on montre que la probabilité de ne pas traverser $\mathcal{L}$ en $2M$ transitions est au plus $(1-p)^2$, et ainsi de suite. La probabilité, partant de x_0, suivant $\mathcal{O}$, de ne pas traverser $\mathcal{L}$ en m étapes tend donc vers 0 quand m tend vers $+\infty$. Autrement dit : $\forall x \in X,\ lim_{m\to\infty}\ \mathbb{P}(x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^{\leq m}\mathcal{L}) = 1$.

Il s'ensuit que pour tout ordonnancement $\mathcal{O}$: $\forall x\ \ \mathbb{P}(x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^{*}\mathcal{L}) = 1$. □

Remarque : Le théorème 4.4 peut être vu comme un cas particulier du théorème 1 de [BDLGJ02] (initialement présenté dans [BGJ99b]).

Ce théorème énonce que, si pour un certain ordonnancement $\mathcal{O}$ l'algorithme considéré converge avec probabilité 1 vers un certain ensemble E_1 de configurations, et s'il existe un entier n et une constante $\delta > 0$ tels que, à partir de toute configuration de E_1, on peut atteindre, en suivant les choix de l'ordonnancement $\mathcal{O}$, une configuration de E_2 en au plus n étapes et avec une probabilité au moins égale à δ, alors l'algorithme suivant $\mathcal{O}$ converge avec probabilité 1 vers l'ensemble E_2.

L'article affirme ensuite que si le résultat est vrai pour tout ordonnancement, alors l'algorithme converge avec probabilité 1 vers l'ensemble E_2.

Notre théorème 4.4 peut donc être vu comme un cas particulier du résultat de [BDLGJ02]. En effet, comme pour chaque ordonnancement la fonction décroît à chaque étape avec probabilité non nulle sauf quand on arrive dans $\mathcal{L}$, on peut utiliser le résultat de [BDLGJ02] pour en déduire qu'on converge avec probabilité 1 vers un ensemble composé de configurations de valeur strictement plus petite et de configurations de $\mathcal{L}$. En itérant cet argument on en déduit la convergence vers $\mathcal{L}$.

Nous voyons cependant un avantage à notre méthode : le mérite de Prop est de proposer une condition concrète et pratique plutôt qu'un critère plus général mais plus abstrait, et d'avoir (espérons-nous) identifié et circonscrit l'essentiel de la difficulté de la preuve de convergence. En effet, c'est cette décroissance en une étape qui va nous permettre de vérifier plus facilement que la fonction trouvée satisfait Prop. Il suffit d'appliquer une fois les règles et de voir comment évolue la fonction.
C'est en ce sens que nous considérons notre critère plus “simple” et plus facilement vérifiable sur des exemples.

Nous avons étudié un certain nombre d'algorithmes en utilisant cette technique de preuve. Nous présentons ci-dessous trois algorithmes, conçus respectivement par Israeli et Jalfon ([IJ90]), Beauquier Gradinariu et Johnen ([BGJ99a]), et enfin Kakugawa et Yamashita ([KY97]), ainsi que leur preuve de convergence.

4.2.1 Convergence de l'algorithme d'Israeli et Jalfon

Dans l'article [IJ90], Israeli et Jalfon présentent un algorithme uniforme d'exclusion mutuelle auto-stabilisante sur un anneau, qui fonctionne même dans le cas où l'ordonnancement n'est pas équitable. Le principe pour permettre la convergence dans le cas d'un ordonnancement arbitraire est ici de permettre aux jetons de se déplacer dans les deux directions : à chaque fois qu'un jeton est sélectionné, il se déplace dans sa direction courante, et au bout de F déplacements il change de direction aléatoirement. Nous allons considérer ici le cas où le paramètre F vaut 1 et donc où le jeton choisit à chaque étape sa direction aléatoirement.

Cet algorithme est différent des autres algorithmes étudiés dans cette thèse, du fait que, comme un jeton peut se déplacer dans les deux sens, l'algorithme est défini par des règles de réécriture à trois lettres, qui peuvent modifier simultanément plusieurs lettres. Chaque processus peut donc lire l'état de ses voisins, et chaque transition peut modifier simultanément l'état de deux processus consécutifs. Cette extension à la modification de plusieurs états simultanément n'introduit pas de conflit car, comme l'ordonnancement est centralisé, un seul processus est sélectionné à chaque étape.

Le système de réécriture est défini par l'ensemble de règles suivant :

$$\begin{array}{lcl} 111 & \to & 101 \\ 011 & \to & \left\{ \begin{array}{l} 101 \text{ avec probabilité } 1/2 \\ 001 \text{ avec probabilité } 1/2 \end{array} \right. \\ 110 & \to & \left\{ \begin{array}{l} 101 \text{ avec probabilité } 1/2 \\ 100 \text{ avec probabilité } 1/2 \end{array} \right. \\ 010 & \to & \left\{ \begin{array}{l} 100 \text{ avec probabilité } 1/2 \\ 001 \text{ avec probabilité } 1/2 \end{array} \right. \end{array}$$

Étant donnée une configuration x, on dit qu'il y a un jeton à la position i si $q_i = 1$. La fonction D_j compte le nombre de jetons dans une configuration x, à savoir $D_j(x) = card(\{i \in \{0, \cdots, N-1\} \mid q_i = 1\})$. Dans cet algorithme, les règles prennent en compte l'état de trois processus consécutifs et illustrent le fait qu'un jeton va se déplacer vers sa droite ou vers sa gauche. Ainsi, nous allons considérer qu'une règle mettant en jeu les processus en position $i-1$, i et $i+1$ va être appliquée à la position centrale, *i.e.* i. Comme on veut vérifier une propriété d'exclusion mutuelle (l'existence au final d'un seul jeton), on va se concentrer sur des configurations initiales comportant au moins un jeton ($D_j(x_0) > 0$). Comme toute règle comporte au moins un jeton dans sa partie droite, si x_0 a au moins un jeton, alors c'est le cas de toutes les configurations qui suivent, et le système n'a pas de blocage.
Il est également clair, en regardant les règles, qu'aucun jeton ne peut être créé, et la fonction D_j est décroissante le long de toute exécution. L'ensemble $\mathcal{L}$ des configurations légitimes est défini comme l'ensemble des configurations contenant exactement un jeton ($D_j = 1$).

Étant donnée une configuration avec un nombre fixé de jetons, mettons $k > 1$, on considère la fonction D_{dist} qui associe à cette configuration x le k-uplet, classé par ordre croissant, des distances entre deux jetons consécutifs. Par exemple pour la configuration $x = 000110001010$, on a $k = 4$ et $D_{dist}(x) = (1, 2, 4, 5)$.

Montrons que, pour tout $x \notin \mathcal{L}$ (contenant au moins deux jetons), et tout choix possible de l'ordonnancement (c'est-à-dire toute position dans x d'un processus dans l'état 1), il existe une stratégie de réécriture qui fait décroître $D = (D_j, D_{dist})$ pour l'ordre lexicographique $\ll$.

Il y a quatre cas à considérer :

- Si $q_{i-1}q_iq_{i+1} = 111$, alors x est de la forme $u111v$ et $x \xrightarrow{i} y = u101v$ avec $D_j(y) = D_j(x) - 1$.
- Si $q_{i-1}q_iq_{i+1} = 011$, alors x est de la forme $u011v$. Notre stratégie consiste à réécrire $x \xrightarrow{i} y = u001v$ avec $D_j(y) = D_j(x) - 1$.
- Si $q_{i-1}q_iq_{i+1} = 110$, alors x est de la forme $u110v$. Notre stratégie consiste à appliquer la transition $x \xrightarrow{i} y = u100v$ ce qui donne $D_j(y) = D_j(x) - 1$.
- Si $q_{i-1}q_iq_{i+1} = 010$, alors x est de la forme $u010v$. Soit g (resp. d) la distance entre le jeton à la position i et le plus proche jeton à sa gauche (resp. à sa droite). Un tel plus proche jeton existe car $D_j(x) > 1$ (ces deux plus proche jetons coïncident lorsque $D_j(x) = 2$). Notre stratégie de réécriture est alors la suivante :
  - si $g \leq d$ on applique $x \xrightarrow{i} y = u100v$,
  - sinon on applique $x \xrightarrow{i} y = u001v$.

  Dans les deux cas on fait décroître la plus petite des deux distances et augmenter l'autre. Comme les distances sont ordonnées par ordre croissant dans D_{dist}, on obtient dans chaque cas $D_{dist}(y) < D_{dist}(x)$.

Dans tous les cas, la fonction D diminue. D'après le théorème 4.4 (appliqué uniquement pour les ordonnancements centralisés), on en conclut que, pour tout ordonnancement centralisé $\mathcal{O}$, $\forall x\ \mathbb{P}(x \xrightarrow{\mathcal{O}}{}^* \mathcal{L}) = 1$. L'algorithme converge donc bien vers l'ensemble des configurations à un jeton.

4.2.2 Convergence de l'algorithme de Beauquier, Gradinariu et Johnen

Dans l'article [BGJ99a], Beauquier, Gradinariu et Johnen présentent un algorithme probabiliste de circulation unidirectionnelle de jeton qui assure la convergence vers l'ensemble des configurations à un jeton probabiliste, et ce quel que soit l'ordonnancement.

Cet algorithme fonctionne quelle que soit la taille N de l'anneau et requiert $PPND(N)^2$ états, où $PPND(k)$ est le plus petit entier non diviseur de k. Ce choix du nombre d'états sert, comme dans l'algorithme de Herman de l'exemple 3.20, à assurer qu'il y a à tout moment un jeton dans une configuration

L'état d'un processus est un couple (d, p), où d est un "état déterministe" et p un "état probabiliste", avec $d,\ p \in \{0, ..., PPND(N) - 1\}$. Le système de transitions $\xrightarrow[BGJ]{}$ est défini par l'ensemble de règles suivant :

$$\begin{array}{lcl} (d,p)(\mathbf{d'},\mathbf{p+1}) & \rightarrow & (d,p)(\mathbf{d+1},\mathbf{p+1}) \\ (d,p)(\mathbf{d'},\mathbf{p'}) & \rightarrow & \left\{ \begin{array}{l} (d,p)(\mathbf{d+1},\mathbf{p'}) \text{ avec probabilité } 1/2 \\ (d,p)(\mathbf{d+1},\mathbf{p+1}) \text{ avec probabilité } 1/2 \end{array} \right. \end{array}$$

où d' (resp. p') est un état déterministe (resp. probabiliste) différent de $d + 1$ (resp. p+1).

Étant donnée une configuration x, soit (d_i, p_i) l'état du processus à la position i dans x. On dit qu'il y a un jeton déterministe (resp. un jeton probabiliste) à la position i si $d_i - d_{i-1} \neq 1$ (resp. $p_i - p_{i-1} \neq 1$). Si l'on étudie les deux règles possibles, on voit que les positions activables sont exactement celles possédant un jeton déterministe.

Considérons le cas où $N = 5$. Dans ce cas $PPND(5) = 2$ et les états probabilistes et déterministes sont à valeurs dans $\{0, 1\}$. Dans la configuration $x = (0,1)(1,0)(1,0)(0,1)(1,1)$, la seule position activable[1] est 2 ($d_2 = d_1 = 1$) donc c'est elle qui va être réécrite. Comme il y a un jeton probabiliste également en position 2, on applique la deuxième règle qui est probabiliste et, selon le résultat du lancer aléatoire, on a $x \xrightarrow[BGJ]{} y_1 = (0,1)(1,0)(0,0)(0,1)(1,1)$ avec probabilité $1/2$, et $x \xrightarrow[BGJ]{} y_2 = (0,1)(1,0)(0,1)(0,1)(1,1)$ avec probabilité $1/2$. Dans les deux cas, la position activable s'est déplacée d'un pas vers la droite.

L'ensemble $\mathcal{L}$ des configurations légitimes est défini comme l'ensemble des configurations possédant un seul jeton probabiliste. On considère comme dans [BGJ99a] le cas d'un ordonnancement arbitraire. Comme le nombre de processus est impair, il y a toujours au moins un jeton déterministe dans une configuration et le système n'a pas de blocage.

Pour prouver la convergence, nous allons considérer la fonction $D = (D_j, D_p, D_d)$ où

- D_j compte le nombre de jetons probabilistes de x
- D_p est la distance minimale entre deux jetons probabilistes de x
- D_d est construite comme suit. Soit $T = \{t_1, t_2, ..., t_k\}$ (où $k = D_j(x)$) l'ensemble des positions des jetons probabilistes ($p_{t_i} - p_{t_i - 1} \neq 1$ pour $1 \leq i \leq k$).
  Comme dans l'algorithme d'Herman, le but est de diminuer la distance minimale entre deux jetons probabilistes, or ici on ne peut réécrire que les indices de jetons déterministes. On va donc considérer une composante de la fonction qui compte le nombre maximal de transitions possibles avant de réécrire un jeton probabiliste "intéressant", c'est-à-dire un jeton probabiliste à distance minimale de son homologue de droite.
  On va donc considérer T' l'ensemble des indices t_k de jetons de T tels que la distance avec le jeton probabiliste à leur droite est minimale (à savoir $t_{k+1} - t_k = D_p(x)$ modulo N). Alors on prend pour D_d la somme sur tous les indices i de jetons déterministes, de la distance d_i entre ce jeton déterministe et le plus proche indice de T' à sa droite. Formellement :

$$D_d(x) = \sum_{\substack{i/\text{ processus } i \text{ possède} \\ \text{un jeton déterministe}}} \min_{j \in T'}((j - i) \text{ modulo } N)$$

[1] rappelons que les positions sont numérotées à partir de 0

Ces trois composantes décrivent les caractéristiques importantes d'une configuration : tout d'abord le nombre de jetons probabilistes (si $D_j = 1$, la configuration est légitime), ensuite la distance minimale entre deux jetons probabilistes (plus ils sont proches, plus ils ont de chances d'entrer en collision et donc de disparaître), et enfin la distance entre les jetons déterministes et les jetons probabilistes susceptibles de faire décroître D_p.

Par exemple dans la configuration $x = (0,0)(1,0)(1,1)(0,1)(1,0)(0,0)(1,1)$ où $N = 7$ (et donc toujours $PPND(N) = 2$), il y a un jeton déterministe d'indice 2, et l'ensemble T des indices de jetons probabilistes est $\{1,3,5\}$. Il est plus facile de voir les jetons si l'on décompose la configuration x en une configuration déterministe $x_d = 0110101$ composée des états déterministes, et une configuration probabiliste $x_p = 0011001$. La distance minimale D_p est 2 et $T' = \{1,3\}$. Comme il n'y a qu'un seul jeton déterministe et qu'il est en position 2, $D_d = \min_{j\in\{1,3\}}((j-2) \text{ modulo } N) = 1$.

Comme l'ordonnancement est arbitraire, on va pouvoir appliquer simultanément des règles à un sous-ensemble quelconque J de processus activables. Comme ces règles ne modifient l'état (probabiliste et déterministe) que d'un seul processus, il n'y a pas de conflit. La stratégie probabiliste que nous appliquons est la suivante, suivant la situation de $i \in J$:

- Si $p_i = p_{i-1} + 1$, il n'y a pas de jeton probabiliste à la position i et on applique la première règle qui est déterministe.
- Si $p_i - p_{i-1} \neq 1$ c'est qu'il y a un jeton probabiliste à la position i et nous choisissons le résultat de la transition probabiliste en suivant la stratégie de réécriture suivante :
  - Si i appartient à T' et s'il est le plus petit indice de J dans T', on change l'état probabiliste p_i en $p_{i-1} + 1$[2].
  - Sinon (lorsque l'indice i n'est pas dans T' ou qu'il existe un autre indice plus petit dans $J \cap T'$) on ne change pas l'état probabiliste p_i.

Il est important de remarquer que lorsqu'on réécrit un jeton, probabiliste ou déterministe, il se décale vers la droite, fusionnant éventuellement avec son voisin de droite. Par exemple on prend $N = 6$, on a $PPND(N) = 4$, et on considère la configuration suivante : $x = (0,0)(1,1)(2,3)(1,0)(2,0)(0,1)$. Intéressons-nous plus précisément aux états déterministes (le cas des états probabilistes est similaire) : $x_d = 012032$. Il y a quatre jetons déterministes en positions 0, 3, 4 et 5.

Si on réécrit le jeton en position 0 on obtient $y_d = 312032$ avec des jetons en positions 1, 3, 4 et 5. Le jeton réécrit s'est bien déplacé d'un pas vers la droite.

Si, encore à partir de x_d on réécrit le jeton à la position 3 on obtient $y'_d = 012332$ qui n'a plus que trois jetons aux positions 0, 4 et 5. Deux jetons ont fusionné et l'un des deux a disparu.

Et enfin si, toujours à partir de x_d, on réécrit le jeton à la position 4, on obtient $y''_d = 012012$, on n'a alors plus que deux jetons en positions 0 et 3. Dans ce cas deux jetons ont fusionné et ont disparu tous les deux. Ceci est dû au fait qu'on avait $d_5 - d_3 = 2$, et donc quand d_4 se change en $d_3 + 1$, les deux jetons disparaissent simultanément.

Nous allons à présent montrer que pour tout $x \notin \mathcal{L}$, c'est-à-dire toute configuration avec au moins deux jetons, pour toute

position activable i sélectionnée par l'ordonnancement, il existe une configuration y telle que $x \xrightarrow[BGJ]{i} y$ avec $D(y) \ll D(x)$, où $\ll$ est l'ordre lexicographique.

[2]On ne déplace donc qu'un seul jeton probabiliste à distance minimale. En effet, si on les déplaçait tous, on risquerait (lorsque tous les jetons sont à distance minimale de leur voisin) de décaler tous les jetons et donc de ne pas diminuer la distance minimale.

- Si un état probabiliste est réécrit dans x, d'après la stratégie ci-dessus il est le seul et il correspond à un jeton probabiliste dont la distance avec son voisin de droite est minimale ($=D_p(x)$). Dans ce cas, ce jeton probabiliste se déplace d'un pas vers la droite et soit $D_p(x)$ diminue, soit le jeton fusionne avec un autre jeton probabiliste et disparaît ($D_j(x)$ diminue).
- Si aucun état probabiliste n'est réécrit, cela signifie que pour tout $i \in J$, i n'appartient pas à T' et donc $d_i \geq 1$. Comme on décale tous les jetons déterministes qui étaient à ces positions, les mesures d_i associées diminuent, et éventuellement certains jetons déterministes fusionnent ce qui fait également décroître D_d.

Dans tous les cas la distance D diminue, et ainsi la propriété Prop du théorème 4.4 est vraie. On en déduit, en appliquant le théorème 4.4 que, quels que soient la configuration initiale et l'ordonnancement, une configuration de $\mathcal{L}$ (avec un seul jeton probabiliste) sera atteinte en un temps fini avec probabilité 1, autrement dit :

$$\forall \mathcal{O},\ \forall x\ \ \mathbb{P}(x \xrightarrow[BGJ]{\mathcal{O}}{}^{*} \mathcal{L}) = 1.$$

On peut noter que, une fois qu'on a atteint une configuration de $\mathcal{L}$, toute la suite de l'exécution reste dans $\mathcal{L}$ qui est clos pour $\xrightarrow[BGJ]{}$. On a donc auto-stabilisation vers $\mathcal{L}$, qui était la propriété recherchée par les auteurs.

Beauquier Cordier et Delaët avaient déjà conçu (dans [BCD95], voir aussi [Del95]) un algorithme optimal d'exclusion mutuelle dans le cas unidirectionnel, mais celui-ci ne fonctionnait que lorsque l'ordonnancement était équitable . Ici ils ne supposent plus l'équité de l'ordonnancement mais l'idée est d'introduire un deuxième type de jetons : des jetons déterministes qui servent à assurer une certaine équité sur les exécutions.

4.2.3 Convergence de l'algorithme de Kakugawa et Yamashita

L'algorithme que nous allons présenter est dû à Kakugawa et Yamashita ([KY97]). Il est unidirectionnel et assure, en présence d'un ordonnancement centralisé quelconque, l'auto-stabilisation vers un ensemble de configurations assurant l'exclusion mutuelle. Ici on n'utilise pas comme dans l'algorithme précédent deux types de jetons (déterministe et probabiliste) mais des *segments* et pour chaque état un *bit aléatoire*. Comme dans l'algorithme précédent l'ordonnancement n'a pas besoin d'être équitable, et l'auto-stabilisation est assurée quelle que soit la taille N de l'anneau.

Dans cet exemple, nous considérons un anneau de N processus $P_0, P_1, ..., P_{N-1}$. L'état d'un processus P_i est le couple $q_i = (l_i, r_i)$ où l_i appartenant à l'ensemble $\{0, 1, ..., N-2\}$ est appelé l'*étiquette* de P_i, et $r_i \in \{0, 1\}$ est un *bit aléatoire*.

Pour cet exemple, la description des actions possibles est plus aisée sous forme d'actions gardées (comme décrites au chapitre 3), et on a besoin des prédicats suivants :

$$\begin{array}{rcl} A_i & = & (l_i \neq l_{i-1} + 1) \wedge ((l_i \neq 0) \vee (r_{i-1} \leq r_i) \vee (l_{i-1} = l_i = 0)) \\ \alpha_i & = & (l_{i-1} = N - 2) \\ C_i & = & (l_i = l_{i-1} + 1) \wedge (l_i \neq 0) \wedge (r_{i-1} \neq r_i) \end{array}$$

Le système de transition se décrit alors à l'aide des commandes gardées suivantes :

A :	`SI` $A_i \wedge \alpha_i$ `ALORS`	$l_i := l_{i-1} + 1$
		$r_i :=$`RandomBit()`
B :	`SI` $A_i \wedge \neg\alpha_i$ `ALORS`	$l_i := l_{i-1} + 1$
		$r_i := r_{i-1}$
C :	`SI` C_i `ALORS`	$r_i := r_{i-1}$

Dans cet algorithme, un *segment* dans une configuration est un ensemble de processus consécutifs $P_a, P_{a+1}, ..., P_b$ tel que $l_a \neq l_{a-1} + 1$, $l_{b+1} \neq l_b + 1$ et, pour tout $i \in \{a, ..., b-1\}$, $l_{i+1} = l_i + 1$. En d'autres mots, c'est une suite maximale de processus consécutifs d'étiquette croissante de 1 en 1. Le processus P_a (resp. P_b) est appelé la *tête* (resp. la *queue*) du segment. Lorsque deux processus consécutifs n'appartiennent pas au même segment, la différence $l_i - l_{i-1}$ entre leurs étiquettes est appelée *écart*.

Comme annoncé au début de cette section, le but est de montrer que le système va converger vers un ensemble satisfaisant l'exclusion mutuelle. Or ici la notion de segment permet de différencier certains processus par rapport aux autres, par exemple tous ceux qui sont en tête d'un segment. Nous allons donc chercher à prouver que le système converge vers un ensemble de configurations dans lesquelles il n'existe plus qu'un seul segment[3].

Notons $D_s(x)$ le nombre de segments d'une configuration x. En étudiant les règles, on constate que soit elles ne modifient pas les étiquettes (C_i), soit elles enlèvent un processus à un segment (celui de l_i) pour le rajouter au segment auquel appartient l_{i-1}. Ainsi le nombre de segments ne peut augmenter, et peut diminuer si l_i était seul dans son segment. On va donc considérer D_s comme une mesure du type φ comme présenté à la section 4.3, et prouver que, à D_s constante, une autre fonction va décroître, pour une bonne stratégie probabiliste.

Dans la suite, on ne va considérer que des transitions utilisant les actions gardées A et B. C'est suffisant pour prouver la convergence étant donné que, comme prouvé dans [KY97], quand D_s ne décroît pas, on peut appliquer seulement un nombre fini de fois des règles suivant C.

La fonction D_s compte le nombre de segments dans une configuration. On définit $\mathcal{L}$ comme l'ensemble des configurations pour lesquelles $D_s \leq 1$. Le système n'a pas de blocage. En effet on peut tout d'abord remarquer que dans le système il y a toujours au moins un segment car $N-1$ (le nombre d'états) ne divise pas N (le nombre de processus)[4]. On en déduit alors que, si tous les bits aléatoires valent 1, la condition A_i est vérifiée pour toutes les têtes de segments, car on a $(l_i \neq l_{i-1} + 1)$ car le processus est en tête de segment, et $1 = r_{i-1} \leq r_i = 1$, et si un des bits aléatoires vaut 0, alors on peut appliquer la commande C jusqu'à faire arriver ce 0 en queue d'un segment, et alors $0 = r_{i-1} \leq r_i$ est vérifiée et on peut appliquer les commandes A ou B.

Comme la fonction D_s ne peut pas augmenter (un segment n'est jamais créé), on va pouvoir raisonner à D_s constant. Nous allons nous concentrer sur les configurations avec $k \leq N-1$ segments [5]. Dans une telle configuration, il y a toujours au moins un écart non nul (voir [KY97] propriété 2). Nous choisissons arbitrairement un couple (s, t) de segments consécutifs tels que l'écart entre s et t est non nul. Pour cela il suffit de numéroter les processus dans la configuration de départ, et de prendre les segments tels que la tête de s a au départ le

[3]Il est impossible qu'il n'y ait aucun segment car, comme il n'y a que $N-1$ étiquettes et N sommets, il est impossible d'avoir pour tout i (modulo N), $l_{i+1} = l_i + 1$ modulo $N-1$.

[4]sauf dans le cas pathologique où $N = 2$ et où le seul état est 0, que nous ne considérerons pas

[5]Ceci n'est pas restrictif étant donné que, si $k = N$, l'application de n'importe quelle règle fait disparaître un segment, et on arrive donc, après une transition, dans le cas précédent.

plus petit indice possible. Nous sommes alors non plus intéressés uniquement par les segments initiaux s et t mais par les segments "dynamiques" $\underline{s}$ et $\underline{t}$ qui suivent l'évolution de s et t le long de l'exécution.

La fonction D est définie par $(D_s(x), E(x), F(x))$ où :

- $D_s(x)$ compte le nombre de segments, comme annoncé précédemment.
- $F(x) = \sum_{\underline{u} \neq \underline{s}} G_{\underline{s},\underline{u}}$, où $G_{\underline{s},\underline{u}}$ est la distance entre la queue du segment $\underline{u}$ et la queue du segment $\underline{s}$ ($\underline{u} \neq \underline{s}$). Par exemple, si P_i est la queue de $\underline{s}$ et P_j la queue de $\underline{u}$, alors $G_{\underline{s},\underline{u}} = i - j$ (modulo N).
- $E(x)$ dépend de la forme des segments $\underline{s}$ et $\underline{t}$ dans la configuration x comme suit :
  - Si la queue de $\underline{s}$ est dans l'état $(b, 0)$ (voir le cas a de la figure 4.1), alors on pose $E(x) = (N - b - 1, \_, \_, \_)$.
  - Si la queue de $\underline{s}$ est dans l'état $(b, 1)$ et aucun processus de $\underline{t}$ n'est dans l'état $(0, 0)$ (voir figure 4.1, cas b), alors $E(x) = (0, N - b' - 1, l, \_)$, où b' est l'étiquette de la queue de $\underline{t}$ et l la longueur de $\underline{t}$.
  - Si la queue de $\underline{s}$ est dans l'état $(b, 1)$ et un processus P de $\underline{t}$ est dans l'état $(0, 0)$ (voir figure 4.1, cas c), alors $E(x) = (0, 0, 0, l')$, où l' est la distance entre le processus P et la queue de $\underline{s}$.

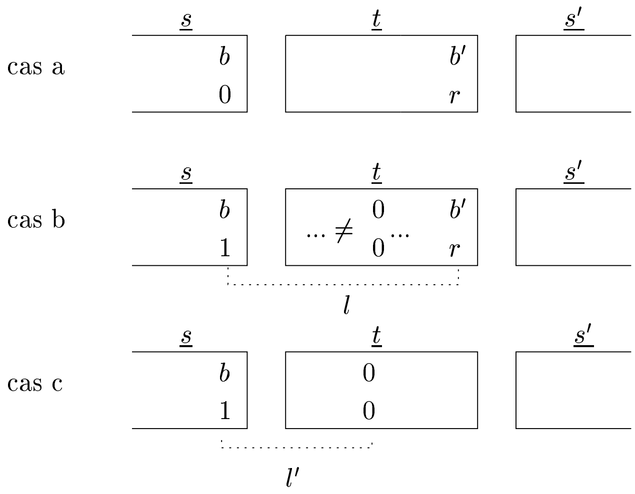

FIG. 4.1 – Configurations possibles

Il reste à montrer que pour cette fonction D et l'ordre lexicographique, on a la propriété Prop et donc d'après le théorème 4.4 l'algorithme de Kakugawa et Yamashita converge vers $\mathcal{L}$.

On va considérer trois cas principaux, selon que la règle est appliquée à la tête du segment $\underline{t}$, à la tête du segment $\underline{s}'$ qui est à droite de $\underline{t}$ ou à la tête d'un autre segment.

1. Si la commande est appliquée à la tête d'un autre segment, on ne modifie ni la queue de $\underline{s}$ ni $\underline{t}$, et donc $E(x)$ reste constante. Par contre la queue d'un segment va se déplacer dans le sens des indices croissants et faire décroître $F(x)$.
2. Si la commande est appliquée au segment dynamique $\underline{s}'$ (qui peut être $\underline{s}$ lorsque la configuration ne contient que deux segments), on a également $F(x)$ qui diminue. Il suffit donc de montrer que $E(x)$ ne croît pas. Si la queue de $\underline{s}$ est dans l'état $(b, 0)$, l'évolution

de $\underline{t}$ ne va pas modifier $E(x)$. Si la queue de $\underline{s}$ est dans l'état $(b, 1)$, et qu'on est dans le cas b de la figure 4.1, alors
- soit on applique la règle B, la queue de $\underline{t}$ passe de (b', r) à $(b' + 1, r)$ et $E(x) = (0, N - b' - 1, l, \_)$ diminue ;
- soit on applique la règle A et on prend pour stratégie de réécriture $r_i := 0$. Le segment $\underline{t}$ contient alors un processus dans l'état $(0, 0)$ et $E(x)$ passe de $(0, N - b' - 1, l, 0)$ à $(0, 0, 0, l')$ donc diminue.

Si la queue de $\underline{s}$ est dans l'état $(b, 1)$, et qu'on est dans le cas c de la figure 4.1, alors l' ne change pas et $E(x)$ non plus.

3. Si on réécrit le segment $\underline{t}$, la queue du segment $\underline{s}$ va changer. Il faut distinguer quatre cas :
   - si la queue de $\underline{s}$ est dans l'état $(b, 0)$ avec $b < N - 2$, elle est changée en $(b + 1, 0)$ et $N - b - 1$ diminue, donc $E(x)$ aussi ;
   - si la queue de $\underline{s}$ est dans l'état $(N - 2, 0)$ alors on va choisir comme stratégie de réécriture de mettre le bit aléatoire à 1. La première composante de $E(x)$ devient nulle et $E(x)$ décroît ;
   - si la queue de $\underline{s}$ est dans l'état $(b, 1)$ et la tête de $\underline{t}$ n'est pas dans l'état $(0, 0)$, alors l ou l' diminue, et $E(x)$ aussi ;
   - enfin si la queue de $\underline{s}$ est dans l'état $(b, 1)$ et la tête de $\underline{t}$ est dans l'état $(0, 0)$, on a ni $r_{i-1} \leq r_i$, ni $l_i \neq 0$, ni $l_i = l_{i-1} = 0$ car on a choisi $\underline{s}$ et $\underline{t}$ spécialement pour avoir un écart non nul. Il n'y a donc pas de règle applicable à la tête de $\underline{t}$.

On a donc vu que, pour toute application d'une commande A ou B dans une configuration avec au moins deux segments, et tant que D_s reste constante, le couple $(E(x), F(x))$ décroît pour l'ordre lexicographique.

Ceci achève notre preuve de convergence de l'algorithme de Kakugawa et Yamashita.

4.3 φ-algorithmes

Pour certains algorithmes, la preuve de convergence peut être simplifiée. En effet, supposons que l'algorithme soit doté d'une fonction φ définie sur les configurations et qui ne croît jamais lors d'une transition. On l'appelle alors φ-algorithme. Dans ce cas on peut ne plus considérer cette composante, et vérifier uniquement que, à φ constante, une certaine fonction D décroît, ce qui simplifie la preuve.

Ceci est bien un cas particulier de tout ce que nous avons vu précédemment dans ce chapitre car, en considérant $D' = (\varphi, D)$ et l'ordre lexicographique, on a bien une fonction comme voulue dans les sections précédentes.

Les résultats des sections précédentes peuvent être reformulés dans ce nouveau cadre, par exemple le théorème 4.4 de la section 4.2 devient

Théorème 4.6. *Étant donnés un système de réécriture $\mathcal{S}$ et une fonction φ non croissante sur les transitions, supposons qu'il existe une fonction D et un ordre $\ll$ tel que*
$Prop_\varphi$: $\forall x \notin \mathcal{L}\ \forall J \subseteq \mathcal{E}(x)\ \exists y :\ (x \xrightarrow[\mathcal{S}]{J} y \wedge (y \in \mathcal{L} \vee \varphi(y) < \varphi(x) \vee D(y) \ll D(x)))$
alors pour tout ordonnancement $\mathcal{O}$, $\forall x$ $\mathbb{P}(x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^* \mathcal{L}) = 1$.

Comme φ ne peut pas croître le long d'une exécution et $x \xrightarrow[\mathcal{S}]{\mathcal{O}} y$, la condition

$\varphi(y) < \varphi(x) \vee D(y) \ll D(x)$ est équivalente à $\varphi(y) < \varphi(x) \vee (\varphi(y) = \varphi(x) \wedge D(y) \ll D(x))$ qui est bien l'ordre lexicographique sur (φ, D)

Pour les exemples considérés, une telle fonction existe, elle correspond au nombre de jetons dans les algorithmes d'Herman (D_1, exemple 4.3, page 69) et d'Israeli et Jalfon (D_j, section 4.2.1), de jetons probabilistes dans l'algorithme de Beauquier, Gradinariu et Johnen (D_j, section 4.2.2), et au nombre de segments dans l'algorithme de Kakugawa et Yamashita (D_s, section 4.2.3). L'existence et la nature de φ sont naturelles étant donné que ces algorithmes servent à assurer la convergence vers un ensemble de configurations où le nombre de jetons (ou segments) est minimal. Les algorithmes sont donc conçus pour que ce nombre n'augmente jamais le long d'une exécution, et il est logique d'utiliser cet argument dans la preuve.

Par contre, dans d'autres algorithmes qui ne sont pas conçus autour d'une certaine valeur décroissante, il n'est pas aisé de trouver une fonction φ appropriée. Par exemple, dans notre variante de l'algorithme du dîner des philosophes probabilistes de Lehmann et Rabin, présentée dans le chapitre 5, nous n'avons pas trouvé de fonction φ intéressante pour simplifier la preuve, c'est pourquoi il n'y a pas de composante de Δ correspondant à ce φ.

On peut remarquer qu'il est toujours possible de trouver, pour chaque algorithme, une fonction φ ne croissant jamais au cours d'une transition : n'importe quelle fonction constante vérifie cette propriété. L'intérêt de φ n'est donc pas dans son existence mais dans l'importance de la simplification qu'elle apporte à la preuve.

L'existence de φ peut avoir un autre avantage : lorsque l'ensemble $\mathcal{L}$ est défini comme l'ensemble des configurations pour lesquelles $\varphi \leq c$ (où c est une constante), on a directement la fermeture de l'ensemble $\mathcal{L}$ pour $\mathcal{S}$. Ainsi la convergence vers $\mathcal{L}$ avec probabilité 1 entraîne directement l'auto-stabilisation probabiliste vers $\mathcal{L}$.

4.4 Calcul du temps moyen de convergence par lumping

Nous revenons dans cette section au cas où l'ordonnancement $\mathcal{O}$ est fixé, déterministe et sans mémoire. Ainsi, le système suivant $\mathcal{O}$ se comporte exactement comme une chaîne de Markov. Nous allons aussi considérer le cas des φ-algorithmes où, comme dans les exemples étudiés, l'ensemble légitime est défini par rapport à φ (i.e. $\mathcal{L} = \{x \in X \mid \varphi(x) \leq c\}$).

Dans le cas où l'algorithme converge avec probabilité 1 vers l'ensemble $\mathcal{L}$, on peut s'intéresser à savoir à quelle vitesse il s'en rapproche, et en combien d'étapes (ou transitions), en moyenne, les exécutions vont arriver dans $\mathcal{L}$. C'est pour connaître cette valeur que nous allons calculer le temps moyen de convergence.

Notations : Soit k un entier strictement plus grand que c, X_k l'ensemble des configurations x vérifiant $\varphi(x) = k$, et $X_{<k}$ l'ensemble des configurations x pour lesquelles $\varphi(x) < k$. Soit X_k^d le sous-ensemble de X_k contenant les configurations pour lesquelles la fonction D vaut d, et Δ_k l'image de X_k par D. On a :

$X_k = \{x \in X \mid \varphi(x) = k\}$.
$X_{<k} = \{x \in X \mid \varphi(x) < k\}$.
$X_k^d = \{x \in X \mid \varphi(x) = k \wedge D(x) = d\}$.
$\Delta_k = \{d \in D(X) \mid \exists x \in X_k :\ D(x) = d\}$.

On remarque que $X_{<k}$ est fermé pour $\mathcal{S}$ (du fait que φ n'augmente jamais le long d'une exécution). On suppose de plus que $\forall x \in X_k\ \exists y \in X_{<k} :\quad x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^* y$. Cela signifie que, pour

toute configuration x telle que $\varphi(x) = k > c$, il est possible de faire décroître φ le long d'une exécution finie. La propriété Prop du théorème 4.2 est une condition suffisante pour garantir l'existence d'une telle exécution.

Ce qui nous intéresse à présent c'est d'obtenir une information quantitative sur le temps moyen de $\mathcal{S}$ suivant $\mathcal{O}$, partant de $x_0 \in X_k$ pour arriver à un état de $X_{<k}$.

Formellement, étant donné $x_0 \in X_k$, soit $T^{X_{<k}}$ l'arbre des exécutions obtenu à partir de $T(\mathcal{O}, x_0)$ (défini à la section 2.3.1) en coupant les arêtes partant des sommets correspondant aux configurations de $X_{<k}$. Soit $Feuill(X_{<k})$ l'ensemble des feuilles de $T^{X_{<k}}$ (c'est à dire les sommets n'ayant aucune arête sortante). Ce qui nous intéresse c'est de calculer (une borne supérieure pour) le temps moyen pour atteindre $X_{<k}$ à partir de x_0, c'est-à-dire $\sum_{v \in Feuill(X_{<k})} \delta(v)\pi(v)$ où $\delta(v)$ et $\pi(v)$ sont respectivement la profondeur et la probabilité de v dans $T^{X_{<k}}$. Ce temps moyen sera noté $E(x_0 \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^* X_{<k})$, ou plus simplement $E(x_0 \xrightarrow[\mathcal{S}]{}{}^* X_{<k})$.

Nous allons à présent expliquer comment calculer cette quantité sous certaines conditions, par "lumping" suivant la fonction D. Dans la théorie de Markov, il est courant de regrouper (*lump* en anglais) des configurations, pour obtenir une chaîne de Markov plus compacte qui donne des informations sur la chaîne originale.

Nous allons montrer ici comment le fait de regrouper des configurations de même valeur pour D, sachant que φ est égale à une constante k peut permettre de calculer
$E(x_0 \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^* Feuill(X_{<k}))$.

Étant donnés $x \in X_k$ et $e \in \Delta_k$, considérons l'expression :

$$\xi(x, e) = \sum_{y \in X_k^e} p(x \xrightarrow[\mathcal{S}]{\mathcal{O}} y).$$

Elle représente la probabilité de passer en une transition *via* $\xrightarrow[\mathcal{S}]{}$ et suivant $\mathcal{O}$ d'un élément x de X_k dans l'ensemble X_k^e (des configurations pour lesquelles D vaut e).
De la même manière considérons l'expression :

$$\xi(x, \perp) = \sum_{y \in X_{<k}} p(x \xrightarrow[\mathcal{S}]{\mathcal{O}} y),$$

où $\perp$ est un nouveau symbole. C'est la probabilité de passer en une transition *via* $\xrightarrow[\mathcal{S}]{}$ suivant $\mathcal{O}$ d'un élément x de x_k dans l'ensemble $X_{<k}$ des configurations pour lesquelles la valeur de φ est strictement plus petite que k.
Enfin, posons $\xi(\perp, \perp) = 1$ pour représenter le fait que $X_{<k}$ est un ensemble dont on ne sort jamais.

Définition 4.7. *On dit qu'un système markovien* $\xrightarrow[\mathcal{S}]{}$ *est* D-lumpable *si, pour tout* $x \in X_k$ *et tout* $e \in \Delta_k \cup \{\perp\}$*, la probabilité* $\xi(x, e)$ *ne dépend que de* $D(x)$*, c'est-à-dire :*

$$\forall d \in \Delta_k \ \forall e \in \Delta_k \cup \{\perp\} \ \forall x, x' \in X_k^d \quad \xi(x, e) = \xi(x', e).$$

On note alors $\xi(d, e)$ *une telle probabilité.*

Étant donné un système de transitions D-lumpable $\xrightarrow[\mathcal{S}]{}$, le *système de transition* D*-lumpé*, noté $\rightsquigarrow$, est défini sur $\Delta_k \cup \{\perp\}$ comme suit :

– pour tout $d, e \in \Delta_k$, $d \rightsquigarrow e$ avec probabilité $\xi(d, e)$,

– pour tout $d \in \Delta_k$, $d \rightsquigarrow \perp$ avec probabilité $\xi(d, \perp)$,
– $\perp \rightsquigarrow \perp$ avec probabilité $\xi(\perp, \perp) = 1$.

Dans ce cas, une transition du type $d \rightsquigarrow e$ est appelée D-transition.

Cette notion de chaîne de Markov lumpable rejoint la définition de bisimulation probabiliste de Larsen et Skou [LS91], en ce sens que le lumping consiste à construire une nouvelle chaîne de Markov, bisimilaire à la première et contenant moins d'états.

D'après la théorie de Markov, si $\xrightarrow[\mathcal{S}]{\mathcal{O}}$ est un système markovien D-lumpable alors le système de transition lumpé $\rightsquigarrow$ est aussi un système markovien. Plus précisément, soit $d \in \Delta_k \cup \{\perp\}$, considérons l'ensemble J_d de tous les couples (e, p) de $(\Delta_k \cup \{\perp\}) \times [0, 1]$ tels que $d \rightsquigarrow e$ avec probabilité p, alors pour tout d, $\Sigma_{(e,p)\in J_d} \ p = 1$.

L'*arbre des exécutions D-lumpé* associé à $\rightsquigarrow$ partant de $d_0 \in \Delta_k \cup \{\perp\}$, noté $U(\mathcal{O}, d_0)$, est tel que :

1. la racine est étiquetée par d_0,
2. tout chemin orienté partant de la racine correspond à une suite possible de D-transitions *via* $\rightsquigarrow$, et
3. chaque sommet w étiqueté par un certain $d \in \Delta_k \cup \perp$ est également étiqueté par la probabilité $\psi(w)$ correspondant au chemin reliant la racine à w dans l'arbre.

On rappelle que si le chemin reliant la racine à w correspond à une exécution de la forme $d_0 \rightsquigarrow d_1 \rightsquigarrow \cdots \rightsquigarrow d_\ell$, alors $\psi(w) = \prod_{j=0}^{\ell-1} \xi(d_j, d_{j+1})$.

Soit $U^\perp$ l'arbre construit à partir de $U(\mathcal{O}, d_0)$ en coupant les arêtes partant des sommets correspondant aux configurations de $\perp$. Soit $Feuill(\perp)$ l'ensemble des feuilles de $U^\perp$. Soient $\psi(w)$ et $\varepsilon(w)$, respectivement la probabilité et la profondeur de w, pour tout sommet w de $U^\perp$. L'expression $\sum_{w\in Feuill(\perp)} \varepsilon(w)\psi(w)$ est le nombre moyen de D-transitions, partant de d_0 pour atteindre $\perp$. On le notera $E_k(d_0 \rightsquigarrow^* \perp)$.

Nous allons maintenant expliquer, en utilisant la théorie de Markov, comment cette quantité peut être calculée et comment elle est reliée à $E(x_0 \underset{\mathcal{S}}{\rightarrow}^* X_{<k})$.

La *matrice de transition D-lumpée* est la matrice carrée de taille $(|\Delta_k|+1)$ dont le coefficient sur la ligne correspondant à d et la colonne correspondant à e est $\xi(d, e)$, pour tous $d, e \in \Delta_k \cup \{\perp\}$.

Si on se rappelle la définition 2.18 de la notion d'états absorbants, il est clair que dans notre cas, $\perp \in \mathcal{A}bs$ (vu que $\perp \rightsquigarrow \perp$ avec probabilité 1). D'autre part, étant donné que $\forall x \in X_k \ \exists y \in X_{<k} : \ x \underset{\mathcal{S}}{\rightarrow}^* y$, et comme la chaîne de Markov est D-lumpable, on a $\forall d \in \Delta_k \ \ d \rightsquigarrow^* \perp$ (convergence vers $\perp$).

La théorie de Markov nous dit dans ce cas, (voir [KS60], p.59) :

Théorème 4.8. (Markov2) *Pour toute chaîne de Markov absorbante, le temps moyen noté $E_k(d \rightsquigarrow^* \mathcal{A}bs)$ pour atteindre les états absorbants est fini. De plus, $(E_k(d \rightsquigarrow^* \mathcal{A}bs))_{d\in\Delta_k} = (\mathcal{I} - \mathcal{Q}_k)^{-1}\mathbf{1}$, où $\mathcal{Q}_k$ est la matrice obtenue en restreignant la matrice de transition D-lumpée à l'ensemble des éléments non absorbants, $\mathcal{I}$ la matrice identité de taille $|\Delta_k|$, et $\mathbf{1}$ est la matrice colonne constituée de $|\Delta_k|$ éléments tous égaux à 1.*

Comme on a restreint la matrice aux éléments non absorbants, la théorie de Markov nous garantit *a priori* que la matrice $(\mathcal{I} - \mathcal{Q}_k)$ est inversible.

D'autre part on a :

Lemme 4.9. *Pour le système de transition D-lumpé, $\mathcal{A}bs = \mathcal{R}ec = \{\perp\}$. Le système de transition D-lumpé se comporte donc comme une chaîne de Markov absorbante.*

Démonstration. Tout d'abord, comme annoncé à la section 2.2.3, $\mathcal{A}bs \subseteq \mathcal{R}ec$ est toujours vrai. Il reste à prouver que $\mathcal{R}ec \subseteq \mathcal{A}bs$ et nous allons le faire par contraposée. Supposons que $d \notin \mathcal{A}bs$. Alors $d \neq \perp$. On a donc $\neg(\perp \rightsquigarrow^* d)$ car $\perp$ est absorbant. Comme on a convergence vers $\perp$, on a $d \rightsquigarrow^* \perp$. Cela donne donc $d \rightsquigarrow^* \perp \wedge \neg(\perp \rightsquigarrow^* d)$ qui caractérise un état transitoire. □

Du théorème 4.8 et du lemme 4.9, on déduit que :

Corollaire 4.10. *Pour tout $d \in \Delta_k$, on a $E_k(d \rightsquigarrow^* \perp) < \infty$. De plus, $(E_k(d \rightsquigarrow^* \perp))_{d\in\Delta_k} = (\mathcal{I} - \mathcal{Q}_k)^{-1}\mathbf{1}$, où $\mathcal{Q}_k$ est la matrice obtenue en restreignant la matrice de transition D-lumpée à l'ensemble des éléments non absorbants (c'est-à-dire en enlevant la ligne et la colonne correspondant à $\perp$).*

De plus, comme le système markovien original $\underset{\mathcal{S}}{\rightarrow}$ est D-lumpable, la théorie de Markov nous dit que le système de transitions D-lumpé $\rightsquigarrow$ satisfait (voir [KS60]) :

$$\forall d \in \Delta_k\ \forall x \in X_k^d \quad E(x \rightarrow^* X_{<k}) = E_k(d \rightsquigarrow^* \perp).$$

Il est intéressant de n'avoir à calculer que $E_k(d \rightsquigarrow^* \perp)$ au lieu de $E(x \underset{\mathcal{S}}{\rightarrow}^* X_{<k})$, étant donné que la matrice de transition de la chaîne de Markov lumpée est beaucoup plus petite que la matrice de la chaîne de Markov originale. Par exemple, pour l'algorithme d'Herman présenté dans les exemples 3.20 et 4.3, dans le cas où k (= le nombre D_j de jetons) vaut 2, la matrice de la chaîne lumpée est de taille $N/2$ alors que la matrice originale est de taille 2^N et que la matrice non lumpée pour deux jetons est de taille N^2. Le calcul de $E_k(d \rightsquigarrow^* \perp)$ dans l'algorithme d'Herman est expliqué dans l'exemple suivant.

4.4.1 Temps moyen de convergence de l'algorithme d'Herman

Montrons tout d'abord que la chaîne de Markov correspondant à l'algorithme d'Herman (voir exemple 3.20, page 62 et sa preuve de convergence à l'exemple 4.3, page 69) est D_2-lumpable pour $k = 2$. On rappelle que $k = D_1$ compte le nombre de jetons, à savoir le nombre de processus dans le même état que leur voisin de gauche, et D_2 compte la distance minimale entre deux jetons consécutifs.

Le fait que la chaîne de Markov est D_2-lumpable tient au fait que la distance minimale est invariante par rotation. Formellement, soit x une configuration avec deux jetons de distance minimale $D_2(x) = d$. Soit i la position du premier jeton et j la position du second. Comme les valeurs 0 et 1 jouent des rôles symétriques, on se moque de la différence entre x et $\bar{x}$, et on peut donc représenter la configuration x par la position de ses deux jetons $x = (i, j)$. La distance d vaut $min(i - j, j - i)$ où $-$ est la soustraction modulo N, et prend ses valeurs dans $\Delta_2 = \{1, 2, \cdots, \lfloor N/2 \rfloor\}$ (comme N est impair, $\lfloor N/2 \rfloor = (N - 1)/2$).

- Si $1 < d < (N - 1)/2$, x se transforme soit en $x_0 = (i, j)$ (cas où aucun jeton ne bouge), $x_1 = (i + 1, j + 1)$ (cas où les deux jetons se déplacent d'un cran vers la droite, $x_2 = (i + 1, j)$ (cas où seul le premier avance) ou $x_3 = (i, j + 1)$ (cas ou seul le deuxième avance), chacun avec une égale probabilité 1/4.
  Par conséquent, $\xi(x, d) = 1/2$ $(= p(x \underset{\mathcal{S}}{\rightarrow} x_0) + p(x \underset{\mathcal{S}}{\rightarrow} x_1))$ et $\xi(x, d + 1) = \xi(x, d - 1) = 1/4$.

- Si $d = 1$, alors $\xi(x,1) = 1/2$, $\xi(x,2) = 1/4$ et $\xi(x,\perp) = 1/4$ (cas où les deux jetons fusionnent).
- Si $d = (N-1)/2$, $\xi(x,d) = 3/4$ et $\xi(x,d-1) = 1/4$.

On constate qu'étant donnés d et e, la valeur de chaque probabilité $\xi(x,e)$ est constante, quel que soit le choix de $x \in X_2^d$, d'où le fait que la chaîne de Markov est lumpable.

Expliquons à présent le calcul de la matrice de transition D-lumpée $\mathcal{Q}_k$ pour $k = 2$ dans l'algorithme d'Herman [Her90]. L'image de l'ensemble des configurations par D_2 est $\Delta_2 = \{1, 2, \cdots, m\}$ avec $m = \lfloor N/2 \rfloor$. Q_2 est la matrice de taille $m \times m$ dont les coefficients sont les $\xi(d,e)$ (où $d, e \in \Delta_2$). Elle est de la forme :

$$\begin{pmatrix} 1/2 & 1/4 & & & & \\ 1/4 & 1/2 & 1/4 & & & \\ & . & . & . & & \\ & & . & . & . & \\ & & & . & . & \\ & & & 1/4 & 1/2 & 1/4 \\ & & & & 1/4 & 3/4 \end{pmatrix}$$

On remarque que le coefficient $\xi(1,\perp) = 1/4$ ne figure pas dans la matrice Q_2 étant donné que la colonne correspondant à $\perp$ a été supprimée.

On peut alors calculer $B_2 = (\mathcal{I} - \mathcal{Q}_2)^{-1}$, ce qui donne :

$$\begin{pmatrix} 4 & 4 & . & . & . & . & 4 \\ 4 & 8 & 8 & . & . & . & 8 \\ . & 8 & 12 & . & . & . & 12 \\ . & . & . & . & & . & . \\ . & . & . & & . & & . \\ . & . & . & & & . & . \\ 4 & 8 & 12 & . & . & . & 4m \end{pmatrix}$$

D'après le corollaire 4.10, on sait que l'application de B_2 à $\mathbf{1}$ donne un vecteur colonne dont les coefficients sont $2d(N-d)$ (pour $d \in \{1, \cdots, m\}$). On a donc

$$E_2(d \rightsquigarrow^* \perp) = 2d(N-d).$$

Le temps moyen de convergence maximum correspond au cas $d = (N-1)/2 = m$ (où les jetons sont le plus loin possible l'un de l'autre), et vaut $2m(m+1) \simeq N^2/2$.

Ceci correspond à $E(x \underset{\mathcal{S}}{\rightarrow}{}^* \mathcal{L})$ pour l'ensemble des configurations à 2 jetons. Nous obtenons ainsi directement grâce à la théorie de Markov le résultat qu'Herman obtient dans son article [Her90] d'une manière moins systématique.

En utilisant ce résultat, Herman explique comment en déduire que le temps moyen de convergence dans le cas général où $x \in X_k$ avec $k > 2$ est $N^2 \lceil \log N \rceil / 2$ (voir [Her90]).

4.4.2 Temps moyen de convergence de l'algorithme d'Israeli et Jalfon

Nous allons présenter ici un calcul d'une borne supérieure du temps moyen de convergence de l'algorithme d'Israeli et Jalfon dans le cas où les configurations ne possèdent que deux jetons.

Rappelons tout d'abord les règles de cet algorithme :

$$
\begin{array}{lcl}
111 & \rightarrow & 101 \\
011 & \rightarrow & \left\{ \begin{array}{l} 101 \text{ avec probabilité } 1/2 \\ 001 \text{ avec probabilité } 1/2 \end{array} \right. \\
110 & \rightarrow & \left\{ \begin{array}{l} 101 \text{ avec probabilité } 1/2 \\ 100 \text{ avec probabilité } 1/2 \end{array} \right. \\
010 & \rightarrow & \left\{ \begin{array}{l} 100 \text{ avec probabilité } 1/2 \\ 001 \text{ avec probabilité } 1/2 \end{array} \right.
\end{array}
$$

Ces règles impliquent trois positions consécutives et en réécrivent au plus deux. Une position possède un jeton si elle correspond à un processus dans l'état 1.

Comme on l'a vu à la section 4.2.1, cet algorithme fonctionne sous un ordonnancement centralisé quelconque, et ne se comporte pas *a priori* comme une chaîne de Markov. Cependant, l'étude du temps moyen de convergence de la section 4.4 ne se fait que dans le cas d'un ordonnancement déterministe et sans mémoire. Nous allons donc utiliser l'astuce suivante :

Dans le cas où une configuration possède seulement deux jetons ($k = 2$), on peut supposer que l'ordonnancement est sans mémoire et sélectionne toujours le même jeton, mettons A, car tout mouvement de l'autre jeton peut être simulé, avec la même probabilité par un mouvement de A.

Il est également facile de voir que la chaîne de Markov correspondant à cet algorithme est lumpable pour la mesure D_{dist} qui qui associe à une configuration x à k jetons le k-uplet, classé par ordre croissant, de distances entre deux jetons consécutifs de x. La chaîne lumpée correspond à une marche aléatoire, et le temps moyen $E_2(d \rightsquigarrow^* \perp)$ correspond au temps moyen pour qu'un jeton à une distance d de l'origine, suivant une marche aléatoire, atteigne cette origine.

Pour $N = 2m + 1$, on a :

$$
\mathcal{Q}_2 = \begin{pmatrix}
0 & 1/2 & & & & \\
1/2 & 0 & 1/2 & & & \\
 & \cdot & \cdot & \cdot & & \\
 & & \cdot & \cdot & \cdot & \\
 & & & \cdot & \cdot & \\
 & & & 1/2 & 0 & 1/2 \\
 & & & & 1/2 & 1/2
\end{pmatrix}
$$

ce qui nous donne :

$$
(\mathcal{I} - \mathcal{Q}_2)^{-1} = \begin{pmatrix}
2 & 2 & . & . & . & . & 2 \\
2 & 4 & . & . & . & . & 4 \\
. & . & 6 & . & . & . & 6 \\
. & . & . & . & & & . \\
. & . & . & & . & & . \\
. & . & . & & & . & . \\
2 & 4 & 6 & . & . & . & 2m
\end{pmatrix}
$$

Le temps moyen $E_2(d \rightsquigarrow^* \perp)$ vaut donc $d(N-d)$ avec un maximum de $m(m+1) \simeq (N/2)^2$ lorsque $d = m$.

En utilisant notre méthode de calcul du temps moyen de convergence, nous retrouvons bien le résultat quadratique donné par Israeli et Jalfon [IJ90].

Chapitre 5

Application au dîner des philosophes

Ce chapitre présente en grande partie le travail publié dans [DFP02]. Il donne une application du critère de convergence donné au chapitre précédent pour montrer une propriété de convergence (que nous appellerons progrès) d'un algorithme de dîner des philosophes probabilistes. Plus précisément, nous montrons qu'une variante de l'algorithme des "free philosophers" de Lehmann et Rabin ([LR81]) assure une propriété de progrès. De plus, l'originalité de notre preuve est qu'elle ne nécessite aucune hypothèse d'équité sur l'ordonnancement, contrairement aux preuves de l'algorithme original des "free philosophers". Pour une motivation de l'étude d'algorithmes sans équité, voir le chapitre 6.

Ce chapitre s'organise comme suit. Dans la section 5.1, nous présentons le problème d'allocation de ressources qu'est le dîner des philosophes, et énonçons le résultat d'impossibilité dans le cas déterministe. Nous décrivons ensuite l'algorithme original du dîner des philosophes probabilistes de Lehmann et Rabin ([LR81]) à la section 5.2. La section 5.3 discute la nécessité (dans l'algorithme original) de l'hypothèse d'équité, et motive l'abandon de cette hypothèse. Nous présentons ensuite notre variante sans équité de l'algorithme du dîner des philosophes probabilistes, en le comparant avec l'algorithme original à la section 5.4. La section 5.5 donne une définition formelle de la propriété de progrès, dans le cas de l'algorithme original puis dans le cas de notre algorithme. La section 5.6 présente la preuve de convergence de notre variante, ainsi que la description précise de la mesure Δ que nous utilisons dans cette preuve.
Après avoir démontré la convergence de cet algorithme, nous nous intéressons à la section 5.7 au temps moyen de convergence, et montrons que pour un ordonnancement malicieux, ce temps peut être au moins exponentiel en le nombre de philosophes présents. Pour finir, nous présentons brièvement un autre algorithme de Lehmann et Rabin, le dîner des philosophes courtois, pour lequel la propriété d'équité est plus importante, et pour lequel il n'est pas possible de l'ôter tout en conservant la propriété de convergence voulue.

5.1 Description du problème

Le problème du dîner des philosophes a été posé pour la première fois par Dijkstra [Dij72] pour résoudre un problème d'allocation de ressources (voir section 3.1.4 pour une définition) et est devenu un paradigme dans le cadre de l'étude des problèmes distribués.

L'idée est la suivante : N philosophes (où N est un paramètre) sont assis en cercle, autour d'une table, avec une baguette entre chaque couple de philosophes voisins. Ils souhaitent déguster un plat de pâtes, mais deux baguettes leur sont nécessaires pour se restaurer (voir figure 5.1).

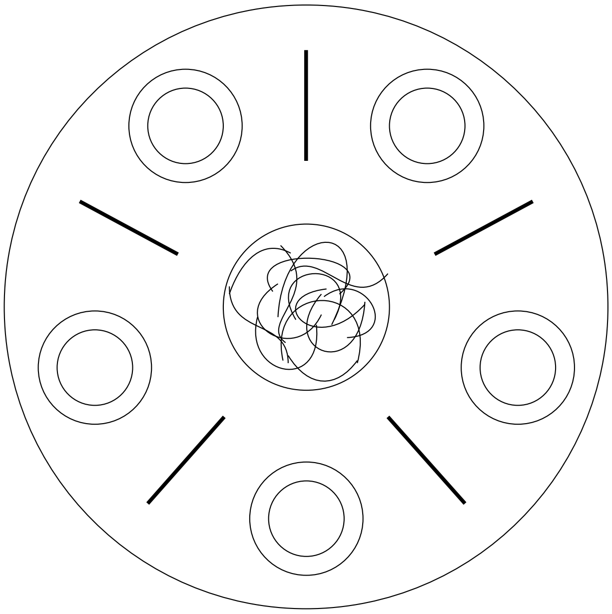

FIG. 5.1 – la table des philosophes pour N=5

Un philosophe peut soit penser (sans aucune interaction avec ses voisins) soit essayer de manger. Pour ce faire, il doit se saisir à la fois de sa baguette de droite et de celle de gauche. Comme chacune des baguettes est partagée avec un voisin, un philosophe ne peut manger que si aucun de ses voisins ne tient l'une des baguettes concernées. Pour représenter ce partage des baguettes, on considère que chaque philosophe partage deux variables en écriture et en lecture, l'une avec son voisin de gauche, l'autre avec son voisin de droite.

L'application d'une règle étant une opération atomique, elle ne peut mettre en oeuvre qu'une seule des deux variables partagées par le philosophe concerné. Un philosophe ne peut par conséquent pas se saisir simultanément des deux baguettes par exemple. Chaque règle consiste donc en un test et une mise à jour (éventuelle) de l'une des deux variables partagées. Le philosophe lit la valeur de la variable partagée et lui donne une nouvelle valeur qui est fonction de l'ancienne valeur et de l'état actuel du philosophe. Il peut aussi changer son état interne appartenant à Σ.

Le problème est de trouver un unique ensemble de règles, valable pour chacun des philosophes qui garantisse la propriété de *progrès* suivante : si à un moment un philosophe a faim, alors un philosophe (mais pas nécessairement le même) va manger un jour, et ce quel que soit l'ordonnancement.

Lehmann et Rabin ont montré dans [RL94] (avec un premier énoncé et une idée de la preuve dans [LR81]) que :

Théorème 5.1. *Il n'y a pas d'algorithme déterministe,* symétrique *et* totalement distribué *assurant la propriété de progrès pour le problème du dîner des philosophes.*

Symétrique signifie ici que les philosophes sont identiques (c'est-à-dire qu'ils suivent tous le même algorithme), et que l'algorithme doit fonctionner même si la configuration initiale est symétrique (toutes les variables partagées sont dans le même état, et tous les philosophes sont dans le même état). Un algorithme est dit totalement distribué si chacun des processus (ici les philosophes) n'a accès qu'à son état propre et à un voisinage restreint de variables partagées (ici l'état des deux baguettes partagées). Il n'existe pas par exemple un processus particulier qui aurait connaissance de l'état de tous les autres processus, ou auquel tous les processus auraient accès.

L'idée de la preuve de l'absence d'une telle solution est que, si l'on considère un ordonnancement qui choisit les philosophes l'un après l'autre le long du cercle, alors en partant d'une configuration symétrique, on peut, à la fin de chaque tour, se retrouver à nouveau dans une configuration symétrique. Cela implique que si lors d'un tour un philosophe réussit à manger, alors à la fin du tour chaque philosophe tiendra ses deux baguettes, ce qui est impossible du fait qu'une baguette ne peut être tenue que par un philosophe à la fois.

Plusieurs solutions déterministes ont été envisagées, "oubliant" l'une ou l'autre des deux contraintes. En voici quelques exemples.

- Dans l'algorithme des "right-left dining philosophers" [Lyn96], les philosophes sont numérotés et le choix de la première baguette dépend du fait que le philosophe possède un numéro pair ou non. S'il est pair, le philosophe essaie de prendre sa baguette de gauche en premier, s'il est impair c'est l'inverse.
- Dans une autre variante, il existe un ordre sur les baguettes, et chaque philosophe essaie de prendre en premier la baguette ayant la plus grande valeur.
- Il peut également y avoir un acteur supplémentaire, qui contrôle l'attribution des baguettes aux philosophes.

Les deux premiers exemples ci-dessus ne respectent pas la contrainte de symétrie, et quant au dernier, il n'est pas totalement distribué.

Pour obtenir une solution symétrique et totalement distribuée, Lehmann et Rabin ont introduit un choix aléatoire dans l'algorithme suivi par chaque philosophe. Ceci permet d'assurer que, avec probabilité 1, la symétrie de la configuration sera rompue. Leur solution, présentée dans la section 5.2, suppose cependant que l'ordonnancement est équitable.

5.2 L'algorithme de Lehmann et Rabin

Nous présentons dans cette partie l'algorithme des "free philosophers" de Lehmann et Rabin [LR81].

L'ensemble d'états de chaque philosophe est $\Sigma = \{T, H, \overleftarrow{W}, \overrightarrow{W}, \overleftarrow{S}, \overrightarrow{S}, \overleftarrow{D}, \overrightarrow{D}, E, L_1, L_2\}$. L'état T représente un philosophe en train de penser (Thinking), H un philosophe qui a faim (Hungry), $\overleftarrow{W}$ (resp. $\overrightarrow{W}$) un philosophe qui attend (Wait) pour prendre sa baguette de gauche (resp. de droite) et qui essaiera de la prendre la prochaine fois qu'il sera choisi. L'état $\overleftarrow{S}$ (resp. $\overrightarrow{S}$) désigne un philosophe qui tient sa première baguette, à savoir celle de gauche (resp. de droite) et qui espère prendre la seconde (Second), $\overleftarrow{D}$ (resp. $\overrightarrow{D}$) un philosophe qui tient encore la baguette de gauche (resp., de droite) mais qui va la lâcher (Drop) la prochaine fois qu'il est sélectionné. L'état E représente un philosophe en train de manger (Eating) et qui tient donc les deux baguettes. Quand il arrive dans l'état L_1, il a fini de manger, tient encore les deux baguettes mais a décidé de reposer la première (disons la gauche). Dans l'état L_2 il ne tient plus que la baguette de droite, et se prépare à la poser.

Les détails se rapportant aux variables partagées ne seront pas présentés ici. Ainsi, par exemple, si le philosophe P_i est dans l'état $\overleftarrow{S}$ ou P_{i-1} est dans l'état $\overrightarrow{S}$, cela implique que la variable représentant la baguette partagée par ces deux philosophes a une valeur correspondant au fait qu'elle est prise.

Remarque : Ici, comme dans la preuve de convergence, nous utilisons une numérotation des processus pour pouvoir les désigner précisément. Ceci ne contredit pas l'hypothèse de symétrie, car cette numérotation n'est pas connue des philosophes eux-mêmes, et donc pas prise en compte dans l'algorithme.

Avec ce modèle, toutes les configurations de Σ^N ne sont pas possibles, étant donné qu'une baguette ne peut être tenue que par un philosophe à la fois. Plus précisément, on dit qu'une configuration est *acceptable* si elle ne contient aucun facteur du type $\overrightarrow{\alpha}\overleftarrow{\beta}$, où la lettre $\overrightarrow{\alpha}$ appartient à $\{\overrightarrow{S}, \overrightarrow{D}, E, L_1, L_2\}$ et $\overleftarrow{\beta}$ appartient à $\{\overleftarrow{S}, \overleftarrow{D}, E, L_1\}$. Il est facile de voir que l'ensemble des configurations acceptables est fermé pour l'application des règles données ci-après. Désormais, on supposera (implicitement ou non) que la configuration initiale, et donc toutes les configurations qui suivent le long d'une exécution, seront acceptables.

Notons également que, du fait qu'une règle ne peut concerner qu'une seule variable partagée, il est impossible pour un philosophe d'aller directement de l'état $\overrightarrow{S}$ à l'état H, sans passer par $\overrightarrow{D}$: il doit d'abord tester la baguette de gauche et se rendre compte qu'elle est prise, puis dans une autre étape déposer la baguette de droite.

Le système de réécriture $\mathcal{S}$ correspondant est défini par l'ensemble de règles suivant :

`Q0` : $\mathbf{T} \to \mathbf{T}$
`Q1` : $\mathbf{T} \to \mathbf{H}$
`R0` : $\mathbf{H} \to \overleftarrow{\mathbf{W}}$ avec probabilité 1/2
ou $\overrightarrow{\mathbf{W}}$ avec probabilité 1/2.
`R1` : $\neg\overrightarrow{hold}\ \overleftarrow{\mathbf{W}} \to \neg\overrightarrow{hold}\ \overleftarrow{\mathbf{S}}$
`R2` : $\overrightarrow{hold}\ \overleftarrow{\mathbf{W}} \to \overrightarrow{hold}\ \overleftarrow{\mathbf{W}}$
`R3` : $\overrightarrow{\mathbf{W}}\ \neg\overleftarrow{hold} \to \overrightarrow{\mathbf{S}}\ \neg\overleftarrow{hold}$
`R4` : $\overrightarrow{\mathbf{W}}\ \overleftarrow{hold} \to \overrightarrow{\mathbf{W}}\ \overleftarrow{hold}$
`R5` : $\overleftarrow{\mathbf{S}}\ \neg\overleftarrow{hold} \to \mathbf{E}\ \neg\overleftarrow{hold}$
`R6` : $\overleftarrow{\mathbf{S}}\ \overleftarrow{hold} \to \overleftarrow{\mathbf{D}}\ \overleftarrow{hold}$
`R7` : $\neg\overrightarrow{hold}\ \overrightarrow{\mathbf{S}} \to \neg\overrightarrow{hold}\ \mathbf{E}$
`R8` : $\overrightarrow{hold}\ \overrightarrow{\mathbf{S}} \to \overrightarrow{hold}\ \overrightarrow{\mathbf{D}}$
`R9` : $\overleftarrow{\mathbf{D}} \to \mathbf{H}$
`R10` : $\overrightarrow{\mathbf{D}} \to \mathbf{H}$
`R11` : $\mathbf{E} \to \mathbf{L}_1$
`R12` : $\mathbf{L}_1 \to \mathbf{L}_2$
`R13` : $\mathbf{L}_2 \to \mathbf{T}$

où $\overrightarrow{hold}$ (resp. $\overleftarrow{hold}$) désigne n'importe quel état de Σ correspondant à un philosophe tenant sa baguette de droite (resp. de gauche), c'est-à-dire $\overrightarrow{S}$, $\overrightarrow{D}$, E, L_1 ou L_2 (resp. $\overleftarrow{S}$, $\overleftarrow{D}$, E ou L_1), et $\neg\overrightarrow{hold}$ (resp. $\neg\overleftarrow{hold}$) désigne n'importe quel état du complémentaire.

Remarque : Les actions sont ici décrites comme des règles de réécriture prenant en compte l'état d'un processus et celui d'un de ses voisins. Cependant, on peut remarquer que, pour le processus qui n'est pas réécrit, la seule chose qui compte s'est de savoir s'il tient ou non la baguette partagée par les deux processus. Cela revient donc bien à ce qu'on a annoncé précédemment, à savoir que chaque philosophe ne sait de ses voisins que le fait qu'ils tiennent ou non leur baguette commune.

Les règles décrivent le comportement général d'un philosophe. Initialement, il pense, et ce pendant aussi longtemps qu'il veut, pouvant rester dans l'état T même s'il est choisi par l'ordonnancement (`Q0`). Il peut ensuite se rendre compte qu'il a faim (`Q1`), décider aléatoirement et de manière équiprobable quelle baguette il va vouloir prendre en premier (`R0`). Ensuite il persiste dans ce choix (`R2` ou `R4`) jusqu'à ce qu'il puisse prendre la baguette lorsqu'elle est libre (`R1` ou `R3`). Il ne repose cette baguette que s'il se rend compte que la deuxième est déjà prise par son voisin (R6 suivi de R9, ou R8 suivi de R10), auquel cas il se retrouve dans l'état H pour essayer à nouveau d'obtenir les deux baguettes. Si, alors qu'il tient sa première baguette il trouve la deuxième baguette libre, il la prend et mange (R5 ou R7). Lorsqu'il a fini, il sort de la phase de repas (R11), pose la baguette de gauche (R12), puis la droite (R13), retournant ainsi à la phase de réflexion. Ce comportement est représenté sur la figure 5.2 (tirée de [PZ86]).

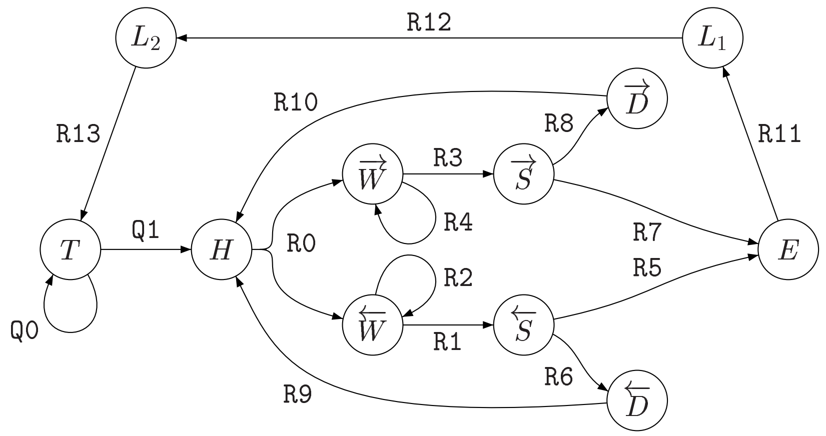

FIG. 5.2 – Illustration de l'algorithme de Lehmann et Rabin.

L'ensemble légitime $\mathcal{L}$ est ici l'ensemble de toutes les configurations (acceptables) de $\Sigma^* E \Sigma^*$, c'est-à-dire les configurations dans lesquelles au moins un philosophe est en train de manger.

5.3 L'hypothèse d'équité

Avec un algorithme comme décrit dans la section 5.2, il est nécessaire d'avoir un ordonnancement équitable (voir définition section 3.5). En effet, supposons qu'un processus soit dans l'état $\overrightarrow{W}$ et son voisin de droite soit dans l'état $\overleftarrow{S}$. Dans ce cas, la baguette commune est prise par celui de droite, et si l'ordonnancement sélectionne toujours le processus de gauche, on va toujours lui appliquer la règle R4 et on va rester pour toujours dans la même configuration (et donc pas satisfaire la propriété de progrès). La condition d'équité est aussi requise du fait de la transition Q0, qui permet à un philosophe de continuer à penser, même s'il est sélectionné par l'ordonnancement. D'un autre côté, la présence des règles R2, R4 et Q0 assure que tous les processus sont activables. Ainsi, pour chaque configuration, on peut appliquer une règle à chaque processus. L'ordonnancement peut donc sélectionner à chaque étape n'importe lequel des processus, du moment que les exécutions engendrées par cet ordonnancement restent équitable.

Cette hypothèse d'équité permet de définir la notion de *tour*, c'est-à-dire une unité de temps pendant laquelle chaque processus est sélectionné (et donc effectue une action) au moins une fois. La preuve de convergence donnée par Pnueli et Zuck [PZ86] utilise très fortement cette notion de tour. La preuve se compose d'une suite de lemmes, dont certains considèrent deux processus consécutifs P_i et P_{i+1}, et ne s'intéressent qu'aux moments où l'on réécrit l'un des deux (ce qui arrive au moins une fois par tour). Ils en déduisent qu'en un nombre fini de tour on arrive avec probabilité 1 dans un ensemble de configurations souhaité.

On peut trouver dans [Lyn96] une autre preuve de l'algorithme de Lehmann et Rabin. L'article [LSS94] présente quant à lui non seulement une preuve mais également le calcul d'une borne supérieure pour le temps moyen de convergence en nombre de tours. Cette borne vaut 63 tours, et ne dépend donc pas du nombre N de philosophes autour de la table.

Une question se pose cependant face à un tel résultat : combien de transitions peuvent correspondre à un tour ? Autrement dit, si le nombre moyen de tours avant qu'un philosophe ne mange est une constante, qu'en est-il du nombre moyen de transitions ?

Dans le cadre présenté ci-dessus, il n'est pas possible d'établir une correspondance entre temps de convergence et nombre de transitions. C'est donc entre autres pour donner un sens à cette correspondance que nous avons considéré une variante de l'algorithme de Lehmann et Rabin, que nous présentons ci-dessous. On peut trouver au chapitre 6 d'autres motivations pour l'étude d'algorithmes sans équité, que ce soit en général ou dans le cas particulier du dîner des philosophes.

5.4 Notre variante

Pour modifier l'algorithme, nous sommes partis de la constatation que les règles `Q0`, `R2` et `R4` sont “invariantes”, dans le sens que leur membre gauche et leur membre droit sont identiques. Quand un philosophe qui est en train de penser (resp. en attente de prendre une première baguette tenue par le voisin concerné) est sélectionné, une transition ne modifiant pas la configuration peut être effectuée. Ceci est représenté sur la figure 5.2 par une boucle sur l'état T (resp. $\overleftarrow{W}$, $\overrightarrow{W}$).

5.4.1 Suppression des règles invariantes

Nous modifions l'algorithme de Lehmann et Rabin principalement en retirant les règles invariantes `Q0`, `R2` et `R4` :

- sans `Q0`, quand un philosophe dans l'état T est sélectionné, on lui applique obligatoirement la règle `Q1` et il passe dans l'état H. Les états T et H jouent alors le même rôle et seront fusionnés dans la suite ;
- sans `R2` (resp.`R4`), lorsqu'un philosophe attend de prendre sa première baguette qui est tenue par son voisin, c'est-à-dire une configuration du genre $\Sigma^*\overrightarrow{W}\overleftarrow{hold}\Sigma^*$ ou $\Sigma^*\overrightarrow{hold}\overleftarrow{W}\Sigma^*$, il n'est plus activable : aucune règle ne peut lui être appliquée. On peut remarquer que ceci diffère de l'algorithme original, dans lequel tous les processus étaient toujours activables.

Étant donné que les états T et H ont fusionné, le nouvel ensemble d'états Σ' est $\Sigma - \{T\}$ et la règle `R13` : $L_2 \to T$ devient `R13'` : $L_2 \to H$. Le système de réécriture $\mathcal{S}$ est changé en $\mathcal{S}' = \mathcal{S} \cup \{\texttt{R13'}\} - \{\texttt{Q0}, \texttt{Q1}, \texttt{R2}, \texttt{R4}, \texttt{R13}\}$. Le comportement d'un philosophe suivant $\mathcal{S}'$ est représenté sur la figure 5.3. Le nouvel ensemble légitime $\mathcal{L}'$ devient pour sa part (l'ensemble des configurations acceptables de) $\Sigma'^*E\Sigma'^*$.

5.4.2 Comparaison avec l'algorithme original

Dans l'algorithme de Lehmann et Rabin, un philosophe peut soit penser (état T), soit essayer de manger (états $H, \overleftarrow{W}, \overrightarrow{W}, \overleftarrow{S}, \overrightarrow{S}, \overleftarrow{D}, \overrightarrow{D}$ et E). Dans notre version de l'algorithme, étant donné que l'état T a été fusionné avec H, le philosophe ne peut qu'essayer de manger.

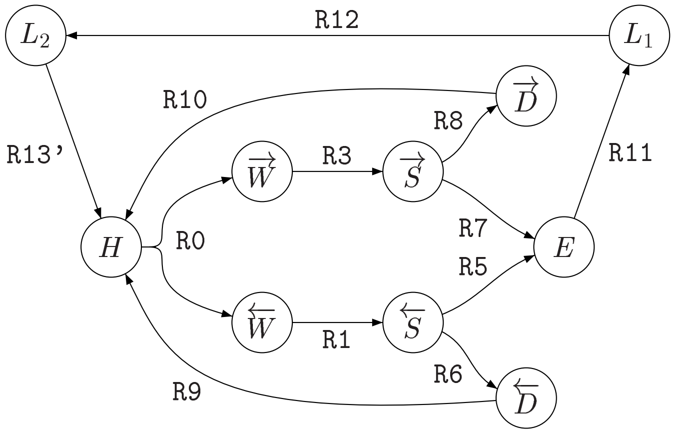

FIG. 5.3 – Illustration de notre variante.

On pourrait croire que cet aspect fait de notre variante une version plus limitée de l'algorithme des “free philosophers”, mais nous allons montrer que ce n'est pas le cas.

Dans notre version, outre le fait de supprimer des règles, nous supprimons aussi l'hypothèse d'équité de l'ordonnancement. Ainsi un philosophe qui se trouve dans l'état H peut être indéfiniment ignoré par l'ordonnancement, et rester à “penser”.
Si au moins un des philosophes n'est pas dans l'état H, alors pour notre algorithme, au moins un de ces philosophes qui n'est pas dans H est activable[1]. L'ordonnancement peut donc indéfiniment ignorer les philosophes dans H, comme il aurait pu le faire pour les philosophes dans T dans l'algorithme original.

Le seul cas où l'ordonnancement ne peut ignorer les philosophes qui “pensent” est le cas où tous sont dans l'état H. À ce moment là on est obligé de considérer que l'un d'entre eux a réellement faim, ce qui est également supposé dans la propriété de progrès de l'algorithme de Lehmann et Rabin (voir section 5.5).

Plus précisément, avec notre algorithme, à toute exécution (infinie et) équitable de l'algorithme original (ne restant pas bloquée dans la configuration T^N) on peut faire correspondre une exécution (infinie) de notre algorithme, où on a remplacé tous les T par des H et où on a juste enlevé les transitions invariantes.
Comme l'exécution de l'algorithme original est équitable et que, tant que la configuration n'est pas T^N, il y a toujours au moins un philosophe susceptible d'effectuer une transition non invariante, l'exécution originale ne contient pas de suite infinie de transitions invariantes, et donc l'exécution obtenue de notre algorithme est également infinie, et “simule” l'exécution originale.

À l'inverse, il existe des exécutions de notre algorithme qui ne peuvent pas correspondre (même en ajoutant des transitions invariantes et en remplaçant des H par des T) à une exécution de l'algorithme original. Ceci est dû au fait que dans notre version de l'algorithme, on permet à l'ordonnancement d'introduire des comportements inéquitables, comme le montre l'exemple suivant :

Exemple 5.2. Considérons l'exécution suivante :

[1]Soit l'un d'entre eux est dans un des états $\overleftarrow{S}$, $\overrightarrow{S}$, $\overleftarrow{D}$, $\overrightarrow{D}$, E, L_1 ou L_2 et donc toujours activable, soit tous sont dans $\overleftarrow{W}$ ou $\overrightarrow{W}$, auquel cas aucune baguette n'est prise et ils sont tous activables.

$H\overleftarrow{W}HHH \rightarrow H\overleftarrow{W}H\overleftarrow{W}H \rightarrow H\overleftarrow{W}H\overleftarrow{S}H \rightarrow H\overleftarrow{W}HEH \rightarrow H\overleftarrow{W}HL_1H \rightarrow H\overleftarrow{W}HL_2H \rightarrow H\overleftarrow{W}HHH \rightarrow H\overleftarrow{W}H\overleftarrow{W}H \rightarrow H\overleftarrow{W}H\overleftarrow{S}H \rightarrow H\overleftarrow{W}HEH \rightarrow H\overleftarrow{W}HL_1H...$
Le long de cette exécution, le deuxième philosophe n'est jamais sélectionné. Si l'ordonnancement était équitable, il finirait par choisir ce philosophe et il faudrait lui appliquer la règle `R1` qui le ferait passer dans l'état $\overleftarrow{S}$, ce qui n'est pas le cas ici. ♣

Ainsi il existe des exécutions qui n'étaient pas prises en compte dans l'algorithme original mais que l'on va prendre en compte ici, tout en conservant la propriété de progrès. C'est en se basant sur ces considérations que nous estimons que notre algorithme conserve l'esprit de l'algorithme original, et n'est pas plus restrictif.

5.5 La propriété de progrès

La propriété de progrès pour l'algorithme de Lehmann et Rabin n'est autre qu'une propriété de convergence probabiliste qui peut être exprimée comme suit :
Progrès : pour tout ordonnancement équitable et centralisé $\mathcal{O}$ et toute configuration $x \in \Sigma^*\{H, \overleftarrow{W}, \overrightarrow{W}, \overleftarrow{S}, \overrightarrow{S}, \overleftarrow{D}, \overrightarrow{D}\}\Sigma^*$, $\mathbb{P}(x \xrightarrow[\mathcal{S}]{\mathcal{O}}{}^* \mathcal{L}) = 1$.

La propriété ci-dessus exprime que, si l'on part d'une configuration dans laquelle au moins un philosophe essaie de manger alors, pour tout ordonnancement équitable, avec probabilité 1, un philosophe (mais pas nécessairement le même) mangera un jour.

Comme on va le montrer dans la section 5.6, pour notre variante $\mathcal{S}'$ de $\mathcal{S}$, la propriété de progrès est satisfaite sans condition d'équité sur l'ordonnancement, à savoir :

Théorème 5.3. *Pour tout ordonnancement centralisé et arbitraire $\mathcal{O}$ et toute configuration $x \in \Sigma'^*$,*

$$\mathbb{P}(x \xrightarrow[\mathcal{S}']{\mathcal{O}}{}^* \mathcal{L}') = 1.$$

Dans le cas de notre variante, ce théorème dit que, quelle que soit la configuration de départ, quel que soit l'ordonnancement centralisé, avec probabilité 1, un philosophe mangera un jour.

Le théorème 5.3 sera prouvé à la section 5.6 en utilisant une variante du théorème 4.4 et en exhibant une fonction appropriée Δ.

Dans cette thèse comme dans tous les travaux effectués (à notre connaissance) sur le problème du dîner des philosophes, nous considérons un ordonnancement centralisé (qui ne sélectionne qu'un processus à la fois). Ceci est dû au fait que l'on ne peut pas permettre à deux processus d'avoir accès simultanément à la même variable partagée. De plus, si l'on ne supposait pas cela, deux philosophes pourraient simultanément se saisir de la même baguette, ce qui serait contradictoire avec le principe même du problème.

Remarque : Dans l'article [RL94], Rabin et Lehmann proposent un relâchement relatif de cette contrainte, disant que l'on peut facilement la modifier pour autoriser que des règles sur différents processus soient effectuées exactement en même temps, dès lors qu'elle n'impliquent pas la même variable partagée. Cette remarque peut tout aussi bien s'appliquer dans notre contexte.
Cependant, ce relâchement est limité et n'augmente pas les propriétés du système. En effet, si plusieurs actions sont effectuées en même temps mais n'implique pas la même variable partagée, alors on peut les appliquer successivement, dans l'ordre que l'on veut, et en obtenant

le même résultat final. L'application de la première d'entre elles (quelle qu'elle soit) ne va pas influencer l'application des suivantes, et ainsi de suite.
Il faut cependant remarquer que, dans le cas général, un système avec un démon centralisé quelconque a un comportement plus restreint que le même système soumis à un démon distribué quelconque (quand il est possible de considérer un tel démon sans risque de conflits).
Ici les comportements sont les mêmes car le relâchement de la contrainte n'est que partiel. Un relâchement total ne serait d'ailleurs pas possible car il induirait des conflits, par exemple si deux philosophes essaient de se saisir de la même baguette en même temps.

Du fait que le relâchement proposé par Lehmann et Rabin n'augmente pas les possibilités du système (en matière de comportements), nous allons nous concentrer uniquement sur le cas des ordonnancements centralisés.

5.6 Preuve de progrès sans équité

Dans cette section, nous allons donner notre preuve de la convergence de notre variante de l'algorithme du dîner des philosophes probabilistes. Tout d'abord, nous donnons les idées sous-jacentes à la preuve et au choix des différentes composantes de notre fonction Δ (section 5.6.1), puis une reformulation du théorème de convergence dans ce nouveau contexte (section 5.6.2). Les motifs permettant de définir les différentes composantes de notre fonction sont ensuite présentés, tout d'abord les jetons (section 5.6.3) puis les anti-jetons (section 5.6.4). Enfin, nous analysons les différents cas de réduction pour prouver que la mesure décroît à chaque étape (section 5.6.5).

Nous allons maintenant exhiber une fonction Δ sur l'ensemble des configurations qui va caractériser, dans un certain sens, la "distance" de la configuration courante x à l'ensemble légitime $\mathcal{L}'$. Nous allons prouver qu'avec une stratégie de réécriture appropriée, la fonction Δ décroît à chaque étape de toute exécution. Plus précisément, nous allons montrer que, pour un certain choix du résultat de la règle probabiliste `R0` (c'est-à-dire soit $\overleftarrow{W}$ soit $\overrightarrow{W}$ selon le contexte de H dans x), et pour toute configuration x non légitime :
- l'application de `R0` fait décroître Δ, et
- l'application de toute autre règle (déterministe) fait décroître Δ ou amène dans $\mathcal{L}'$.

Dans toute la suite, le symbole W représente une lettre de $\{\overleftarrow{W}, \overrightarrow{W}\}$. De même S et D représentent une lettre de $\{\overleftarrow{S}, \overrightarrow{S}\}$ et $\{\overleftarrow{D}, \overrightarrow{D}\}$ respectivement. Nous définissons également les ensembles $\overleftarrow{\Sigma}' = \{H, \overleftarrow{W}, \overleftarrow{S}, \overleftarrow{D}\}$ et $\overrightarrow{\Sigma}' = \{H, \overrightarrow{W}, \overrightarrow{S}, \overrightarrow{D}\}$. (On remarque que $\overleftarrow{\Sigma}' \cap \overrightarrow{\Sigma}' = \{H\}$.)

5.6.1 Idées de la preuve

Cette section présente les idées, souvent informelles, sur lesquelles se base notre preuve, qui sera présentée *in extenso* dans les sections 5.6.3 à 5.6.5.

Comme ce sera précisé à la section 5.6.2, page 99, nous allons nous intéresser tout d'abord au cas où aucun philosophe n'est dans l'état L_1 ou L_2, ce qui est vrai tant qu'aucun philosophe n'a encore mangé. À la fin de la section 5.6.5 nous expliquons comment modifier Δ pour prendre en compte ces deux états.

La définition de notre fonction Δ est basée sur le fait que toute configuration non légitime peut se décomposer de la manière suivante :

- des "jetons", à savoir des facteurs de deux lettres appartenant à $\overleftarrow{\Sigma'}\overrightarrow{\Sigma'}$,
- des "anti-jetons" qui sont des facteurs de deux lettres appartenant à $\overrightarrow{\Sigma'}\overleftarrow{\Sigma'}$,
- et enfin des lettres n'appartenant à aucune des deux catégories ci-dessus. Ces lettres sont dans l'ensemble $\{W, S, D\}$ car tout H appartient soit à un jeton soit à un anti-jeton (voir proposition 5.8 ci-après)

Le terme de jeton n'a pas ici de signification physique particulière. Il sert simplement à désigner une suite de deux lettres ayant un lien particulier décrit ci-dessus. On remarque aussi que, comme on ne considère que des configurations acceptables, certaines configurations de $\overrightarrow{\Sigma'}\overleftarrow{\Sigma'}$ ne sont pas possibles pour un anti-jeton. Pour le jeton on n'a pas ce problème, et toutes les possibilités sont *a priori* possibles, si le contexte le permet.

Dans la suite de la preuve, on va vouloir caractériser les configurations en terme de jetons, d'anti-jetons et de lettres entre les deux. Initialement, on peut se trouver dans une configuration sans jeton, par exemple $\overleftarrow{W}^N$. Dans ce cas, un jeton est créé au plus tard la première fois qu'un H est produit. Plus précisément,

Proposition 5.4. *Si une configuration non légitime x contient un H, alors elle contient au moins un jeton.*

Cette proposition n'est pas nécessaire pour la preuve (car lorsqu'il n'y a pas de jeton, on va également montrer qu'une étape de réécriture fait décroître Δ), mais elle permet de se rendre compte du rôle que joue la topologie en anneau dans cet algorithme.

Démonstration. Considérons les différentes lettres qui peuvent entourer ce H dans x.

- Soit on a un motif appartenant à $H\overrightarrow{\Sigma'}$ ou $\overleftarrow{\Sigma'}H$; dans ce cas le H appartient à un jeton, et la proposition est vérifiée.
- Soit ce H est dans un motif de la forme $\overrightarrow{\alpha}H\overleftarrow{\beta}$ avec $\overrightarrow{\alpha} \in \overrightarrow{\Sigma'} \setminus H$ et $\overleftarrow{\beta} \in \overleftarrow{\Sigma'} \setminus H$. Dans ce cas il n'appartient plus à un jeton, mais $H\overleftarrow{\beta}$ est un motif dans $\overrightarrow{\Sigma'}\overleftarrow{\Sigma'}$. Soit i l'indice de $\overleftarrow{\beta}$ dans x. On va parcourir l'anneau à partir de i dans le sens des indices croissants et noter k le premier indice rencontré tel que la lettre d'indice k soit dans $\overrightarrow{\Sigma'}$. Cet indice k existe car $\overrightarrow{\alpha} \in \overrightarrow{\Sigma'}$, et on a alors, en positions k et $k-1$, un motif appartenant à $\overleftarrow{\Sigma'}\overrightarrow{\Sigma'}$, autrement dit un jeton.

□

Notre stratégie de réécriture vise à préserver les jetons, du moins ceux d'entre eux qui vont être répertoriés dans la liste de jetons π (décrite à la section 5.6.3), et à conserver leur position, c'est-à-dire changer, à l'aide de `R0`, $\mathbf{H}\overrightarrow{\beta}$ en $\overleftarrow{\mathbf{W}}\overrightarrow{\beta}$, et $\overleftarrow{\alpha}\mathbf{H}$ en $\overleftarrow{\alpha}\overrightarrow{\mathbf{W}}$. Avec cette stratégie, une fois créé, un jeton (de π) ne peut pas disparaître.

Un jeton correspond à la situation, décrite dans [RL94], où le dernier choix aléatoire d'un philosophe est la baguette de gauche, alors que le dernier choix aléatoire de son voisin de droite est la baguette de droite. Dans ce cas, après un nombre fini de réécritures dans le jeton utilisant notre stratégie de réécriture, un des deux philosophes va manger. Si l'on supposait que l'ordonnancement est équitable, la preuve du progrès serait presque terminée car l'équité assure que chaque philosophe est sélectionné infiniment souvent le long de chaque exécution (infinie), et avec nos règles tout philosophe sélectionné change d'état.

Comme on ne suppose pas l'équité de l'ordonnancement, il faut encore montrer qu'il n'y a pas de suite infinie de réécriture hors des jetons.

Nous allons nous baser sur le fait que, une fois qu'un jeton a été créé, une configuration quelconque se compose de motifs de la forme $\overleftarrow{\alpha_1}\overrightarrow{\beta_1}uv\overleftarrow{\alpha_2}\overrightarrow{\beta_2}$ avec $u \in \overrightarrow{\Sigma'}^*$ et $v \in \overleftarrow{\Sigma'}^*$.
Les $\overleftarrow{\alpha_i}\overrightarrow{\beta_i}$ sont les jetons, et le seul anti-jeton de ce motif est $\overrightarrow{\gamma}\overleftarrow{\delta}$ où $\overrightarrow{\gamma}$ est la dernière lettre de $\overrightarrow{\beta_1}u$ (qui peut être $\overrightarrow{\beta_1}$ lorsque u est le mot vide) et $\overleftarrow{\delta}$ est la première lettre de $v\overleftarrow{\alpha_2}$ (qui peut être $\overleftarrow{\alpha_2}$ lorsque v est le mot vide).

Nous utiliserons également une fonction Δ qui est un 7-uplet $(\Delta_1, \cdots, \Delta_7)$. Chaque composante est, informellement, une fonction de Σ^N dans $\mathbb{N}$, telle que Δ va décroître pour l'ordre lexicographique à chaque étape de réécriture (sauf si l'on atteint $\mathcal{L}'$). Ceci est démontré à la section 5.6.5 par une analyse systématique des différents cas de réécriture qui peuvent se produire. Les principaux cas de figure sont décrits ci-dessous. Pour les définitions précises des différentes composantes de Δ, se reporter aux sections 5.6.3 à 5.6.5.

Dans la suite de cette idée de la preuve, de nombreux détails sont volontairement omis. Par exemple le cas où un jeton et un anti-jeton se chevauchent n'est pas traité, de même il n'est pas précisé que tous les jetons et les anti-jetons ne sont pas toujours pris en compte. Seuls nous intéressent ceux qui sont répertoriés dans la liste de jetons π et la liste d'anti-jetons ψ. Tous ces détails figurent bien évidemment dans les sections 5.6.3 à 5.6.5.

Considérons tout d'abord le cas où la réécriture a lieu dans un jeton. Comme il a déjà été constaté, grâce à notre stratégie de réécriture, le nombre de jetons (Δ_1) est conservé. D'autre part, la réécriture de D en H, H en W, ou W en S fait décroître Δ_2 (car Δ_2 vérifie $\Delta_2(D) > \Delta_2(H) > \Delta_2(W) > \Delta_2(S)$), alors que la réécriture de S en E donne une configuration de $\mathcal{L}'$. On en déduit que la réécriture au sein d'un jeton fait décroître Δ ou mène dans une configuration légitime.

Nous allons maintenant donner une idée des raisons pour lesquelles la réécriture d'un anti-jeton fait également décroître Δ (ou crée un nouveau jeton). Notre stratégie de réécriture (pour la règle `R0`) pour les anti-jetons vise à les garder dans la même position. Par exemple, un anti-jeton de la forme $\mathbf{H}\overleftarrow{\delta}$ sera changé en $\overrightarrow{\mathbf{W}}\overleftarrow{\delta}$, et $\overrightarrow{\gamma}\mathbf{H}$ en $\overrightarrow{\gamma}\overleftarrow{\mathbf{W}}$.

D'autre part, les anti-jetons sont de deux types :
- non orientés, c'est-à-dire de la forme $H\overleftarrow{W}$, $\overrightarrow{W}\overleftarrow{W}$ ou $\overrightarrow{W}H$
- orientés, soit à gauche donc de la forme $\{\overrightarrow{S}, \overrightarrow{D}\}\overleftarrow{\delta}$, soit à droite, de la forme $\overrightarrow{\gamma}\{\overleftarrow{S}, \overleftarrow{D}\}$.

Le terme d'orienté a été choisi ainsi car, lorsqu'un anti-jeton est orienté, mettons à gauche, il ne peut se déplacer que vers sa gauche.

Un anti-jeton est réécrit au maximum deux fois avant de devenir orienté et donc de faire décroître la composante Δ_4, qui vaut N moins le nombre d'anti-jetons orientés. De plus, un anti-jeton orienté, par exemple à gauche, c'est-à-dire de la forme $\{\overrightarrow{S}, \overrightarrow{D}\}\overleftarrow{\delta}$ peut, lorsqu'il est réécrit,
- soit rester à la même place avec la même orientation, et dans ce cas Δ_6 (la somme des coefficients des lettres des anti-jetons) décroît,
- soit rester à la même position en perdant son orientation, ce qui n'est possible que si Δ_3 (mesure du nombre de motifs du type $\{H, \overrightarrow{W}\}\overrightarrow{D}$ et $\overleftarrow{D}\{H, \overleftarrow{W}\}$) ou Δ_2 décroît,
- soit se déplacer d'une place vers la gauche, en conservant la même orientation, auquel cas Δ_5, la somme des distances des anti-jetons orientés, diminue.

Par exemple, le fait de réécrire $\overrightarrow{\mathbf{S}}\overleftarrow{W}$ en $\overrightarrow{\mathbf{D}}\overleftarrow{W}$ fait décroître Δ_6 (car $\Delta_6(S) > \Delta_6(D)$). D'autre part, le changement de $\lambda\overrightarrow{\mathbf{D}}\overleftarrow{W}$ en $\lambda\mathbf{H}\overleftarrow{W}$ fait diminuer Δ_2, Δ_3 ou Δ_5, selon la nature de la lettre λ à gauche de l'anti-jeton. Si par exemple $\lambda = \overrightarrow{W}$, $\overrightarrow{W}\overrightarrow{\mathbf{D}}\overleftarrow{W}$ se réécrit en $\overrightarrow{W}\mathbf{H}\overleftarrow{W}$, l'anti-jeton a perdu son orientation (donc Δ_4 augmente), mais Δ_3 diminue car un motif $\overrightarrow{W}\overrightarrow{D}$

disparaît ; si on prend $\lambda = \overrightarrow{S}$ alors $\overrightarrow{S}\overrightarrow{\mathbf{D}}\overleftarrow{W}$ se réécrit en $\overrightarrow{S}\mathbf{H}\overleftarrow{W}$, un nouvel anti-jeton $\overrightarrow{S}\mathbf{H}$ apparaît, il est également orienté vers la gauche et Δ_5 décroît. Dans chaque cas, la réécriture d'un anti-jeton fait décroître Δ.

Supposons pour finir que la règle de réécriture est appliquée en dehors des jetons et des anti-jetons. D'après la proposition 5.8 de la section 5.6.4, ces lettres sont du type D, W ou S. Le fait de réécrire D en H crée un nouveau jeton (et donc Δ_1 diminue) ; le fait de réécrire W en S ou S en D fait diminuer Δ_7 (car $\Delta_7(W) > \Delta_7(S) > \Delta_7(D)$) ; enfin le fait de réécrire S en E produit une configuration de $\mathcal{L}'$.

Ainsi, quelle que soit la position réécrite, l'application d'une règle de réécriture dans une configuration non légitime peut soit faire décroître Δ, soit mener dans $\mathcal{L}'$. Ceci termine notre explication informelle de la preuve.

Un exemple d'exécution

Rappelons que notre stratégie de réécriture tend à préserver les jetons et leur position. Dans l'exemple donné à la figure 5.4, il y a deux jetons, mettons J_1 et J_2, qui correspondent aux deux premières lettres et aux deux dernières lettres de la configuration. Entre J_1 et J_2 se trouve un anti-jeton A. Nous allons à présent décrire l'évolution générale de A entre J_1 et J_2. Initialement A est orienté à droite et se déplace en direction de B_2 jusqu'à ce qu'ils se chevauchent (dernière configuration de la première ligne). Il perd alors son orientation, et quelques transitions plus tard (dernière configuration de la deuxième ligne) se retrouve orienté dans l'autre sens. Un tel "demi-tour" nécessite la réécriture de la lettre de gauche de J_2. L'anti-jeton A fait alors des aller-retour entre B_1 et B_2. Le point important est que tout demi-tour implique la réécriture d'un jeton ou la diminution de Δ_3, et donc que si l'on ne réécrit pas dans les jetons, comme Δ_3 ne va pas augmenter, les anti-jetons vont finir par se retrouver bloqués et ne vont plus pouvoir être réécrits sans atteindre $\mathcal{L}'$.

Une suite de réécritures de A est illustré à la figure 5.4.

$$
\begin{array}{l}
\overleftarrow{W}\underset{\Delta_5=3}{\underline{\overrightarrow{W}\overleftarrow{\mathbf{D}}}}\overleftarrow{D}\overleftarrow{D}\overrightarrow{S}
\xrightarrow{\Delta_5}
\overleftarrow{W}\overrightarrow{W}\underset{\Delta_5=2}{\underline{\mathbf{H}\overleftarrow{D}}}\overleftarrow{D}\overrightarrow{S}
\xrightarrow{\Delta_6}
\overleftarrow{W}\overrightarrow{W}\underset{\Delta_5=2}{\underline{\overrightarrow{W}\overleftarrow{\mathbf{D}}}}\overleftarrow{D}\overrightarrow{S}
\xrightarrow{\Delta_5}
\overleftarrow{W}\overrightarrow{W}\overrightarrow{W}\underset{\Delta_5=1}{\underline{H\overleftarrow{\mathbf{D}}}}\overrightarrow{S}
\\
\xrightarrow{\Delta_2}
\overleftarrow{W}\overrightarrow{W}\overrightarrow{W}\underline{H\mathbf{H}}\overrightarrow{S}
\xrightarrow{\Delta_2}
\overleftarrow{W}\overrightarrow{W}\overrightarrow{W}\underline{\mathbf{H}\overleftarrow{W}}\overrightarrow{S}
\xrightarrow{\Delta_6}
\overleftarrow{W}\overrightarrow{W}\overrightarrow{W}\underline{\overrightarrow{\mathbf{W}}\overleftarrow{W}}\overrightarrow{S}
\xrightarrow{\Delta_4}
\overleftarrow{W}\overrightarrow{W}\overrightarrow{\mathbf{W}}\underset{\Delta_5=3}{\underline{\overrightarrow{S}\overleftarrow{W}}}\overrightarrow{S}
\\
\xrightarrow{\Delta_7}
\overleftarrow{W}\overrightarrow{W}\overrightarrow{S}\underset{\Delta_5=3}{\underline{\overrightarrow{\mathbf{S}}\overleftarrow{W}}}\overrightarrow{S}
\xrightarrow{\Delta_6}
\overleftarrow{W}\overrightarrow{W}\overrightarrow{S}\underset{\Delta_5=3}{\underline{\overrightarrow{\mathbf{D}}\overleftarrow{W}}}\overrightarrow{S}
\xrightarrow{\Delta_5}
\overleftarrow{W}\overrightarrow{\mathbf{W}}\underset{\Delta_5=2}{\underline{\overrightarrow{S}H}}\overleftarrow{W}\overrightarrow{S}
\xrightarrow{\Delta_2}
\overleftarrow{W}\overrightarrow{S}\underset{\Delta_5=2}{\underline{\overrightarrow{\mathbf{S}}H}}\overleftarrow{W}\overrightarrow{S}
\\
\xrightarrow{\Delta_6}
\overleftarrow{W}\overrightarrow{S}\underset{\Delta_5=2}{\underline{\overrightarrow{D}\mathbf{H}}}\overleftarrow{W}\overrightarrow{S}
\xrightarrow{\Delta_6}
\overleftarrow{W}\overrightarrow{S}\underset{\Delta_5=2}{\underline{\overrightarrow{\mathbf{D}}\overleftarrow{W}}}\overleftarrow{W}\overrightarrow{S}
\xrightarrow{\Delta_5}
\overleftarrow{W}\underset{\Delta_5=1}{\underline{\overrightarrow{S}\mathbf{H}}}\overleftarrow{W}\overleftarrow{W}\overrightarrow{S}
\xrightarrow{\Delta_6}
\overleftarrow{W}\underset{\Delta_5=1}{\underline{\overrightarrow{S}\overleftarrow{W}}}\overleftarrow{W}\overleftarrow{W}\overrightarrow{S}
\end{array}
$$

FIG. 5.4 – Un exemple d'exécution

Toutes les configurations de cet exemple possèdent deux anti-jetons : un qui se compose de la première et la dernière lettre du mot ($\overrightarrow{S}\overleftarrow{W}$), et l'autre, A, qui est souligné et "étiqueté" par la valeur de Δ_5 lorsqu'il est orienté. La lettre en gras est celle qui va être changée. Sur chaque flèche représentant une transition est précisée la première composante de Δ qui décroît. Dans toute l'exécution, Δ_1 et Δ_3 restent constantes. Dans la dernière configuration, l'anti-jeton $\overrightarrow{S}\overleftarrow{W}$ ne peut pas être réécrit sans atteindre $\mathcal{L}'$.

5.6.2 Le théorème de convergence

Pour prouver la propriété de progrès, nous ne pouvons pas appliquer directement le théorème 4.4, et cette section vise à expliquer pourquoi, ainsi qu'à donner une variante du théorème qui puisse s'appliquer dans ce contexte.

Notons que la lettre H peut, dans une configuration, appartenir à deux jetons qui se chevauchent, par exemple dans le cas $\overleftarrow{W}H\overrightarrow{D}$ (voir section 5.6.3). De même la lettre H peut appartenir à deux anti-jetons qui se chevauchent, comme $\overrightarrow{S}H\overleftarrow{W}$. Pour pouvoir définir Δ malgré de telles ambiguïtés, chaque configuration x sera munie de deux listes : une liste de jetons π, définie à partir de x et une liste d'anti-jetons ψ, définie à partir de x et π.

La fonction Δ sera alors définie pour les triplets (x, π, ψ).
Pour prouver la propriété de progrès, nous allons utiliser une version du théorème 4.4, reformulé comme suit :

Théorème 5.5. *Étant donné un système de réécriture $\mathcal{S}'$, s'il existe une fonction Δ et un ordre $\ll$ tel que*
Prop" : $\forall(x, \pi, \psi)$ *avec* $x \notin \mathcal{L}', \forall i \in \mathcal{E}(x) \quad \exists(x', \pi', \psi')$

$$(x \xrightarrow[\mathcal{S}']{i} x' \wedge (x' \in \mathcal{L}' \ \vee \ \Delta(x', \pi', \psi') \ll \Delta(x, \pi, \psi))),$$

alors, pour tout ordonnancement centralisé $\mathcal{O}$: $\forall x \ \ \mathbb{P}(x \xrightarrow[\mathcal{S}']{\mathcal{O}}{}^* \mathcal{L}') = 1.$

Les sections 5.6.3 à 5.6.5 sont consacrées à la preuve formelle de la propriété Prop" (voir proposition 5.13). Comme la preuve de Prop" ne nécessite que de considérer des configurations non légitimes ($\notin \mathcal{L}'$), nous n'allons donc prendre en compte dans la preuve que des configurations de $(\Sigma' - \{E\})^*$).
Pour des raisons de simplicité, nous allons nous restreindre dans toute la preuve à des configurations qui ne contiennent pas les lettres L_1 et L_2 (obtenues lorsqu'un philosophe, après avoir mangé, repose successivement ses deux baguettes). Ceci n'est pas une restriction tant qu'aucun philosophe n'a encore mangé, et donc quand on s'intéresse au fait de prouver qu'un premier philosophe va manger, un jour. Nous expliquons à la fin de la section 5.6.5 comment cette preuve peut être modifiée pour prendre également en compte les états L_1 et L_2. La nouvelle fonction Δ' possède alors le même nombre de composantes, mais celles-ci, ainsi que la définition des jetons et des anti-jetons, sont quelque peu modifiées.

5.6.3 Jetons et composantes Δ_1, Δ_2, Δ_3

Un *jeton* dans une configuration x est un facteur de x composé de deux lettres dans $\overleftarrow{\Sigma'}\overrightarrow{\Sigma'}$.
L'*indice* d'un jeton $\overleftarrow{\alpha}\overrightarrow{\beta}$ est la position de sa première lettre $\overleftarrow{\alpha}$.
On remarque que, du fait de la présence de la lettre H qui appartient à la fois à $\overleftarrow{\Sigma'}$ et $\overrightarrow{\Sigma'}$, deux jetons peuvent se chevaucher. Par exemple, dans l'expression $\overleftarrow{W}H\overrightarrow{S}$ il y a deux jetons qui se chevauchent : $\overleftarrow{W}H$ et $H\overrightarrow{S}$.

Étant donnée une configuration x, nous allons nous intéresser à une suite π d'indices de jetons *disjoints* de x, i.e. tels que $j \geq i + 2$ pour tous indices consécutifs i et j de π. Nous allons également supposer que π est *maximal*, c'est-à-dire qu'entre deux indices consécutifs i et j de π il n'y a pas de jeton d'indice k avec $i + 2 \leq k \leq j - 2$. Autrement dit, la maximalité signifie qu'il n'existe pas de jetons non répertorié dans π et disjoint de tout jeton de π.

Une suite maximale d'indices de jetons disjoints de x est appelée une *liste de jetons* de x. D'après ce que l'on a dit ci-dessus (du fait que deux jetons peuvent se chevaucher), une telle liste n'est pas unique. L'ensemble $Jeton(\pi)$ est défini comme l'ensemble des lettres de x qui se trouvent à une position ℓ telle que $\ell \in \pi$ ou $\ell - 1 \in \pi$. $Jeton(\pi)$ répertorie donc toutes les positions de processus qui appartiennent à un jeton de π.

Exemple 5.6. La configuration $\overleftarrow{W}\overrightarrow{W}\overleftarrow{S}\overleftarrow{W}H\overrightarrow{D}$, possède deux listes de jetons possibles $\pi_1 = \{0, 3\}$ et $\pi_2 = \{0, 4\}$. Les jetons sont $\overleftarrow{W}\overrightarrow{W}$ et $\overleftarrow{W}H$ pour π_1, $\overleftarrow{W}\overrightarrow{W}$ et $H\overrightarrow{D}$ pour π_2. ♣

Remarque : Il est à noter que, avec notre définition, une suite maximale d'indices de jetons disjoints (= une liste de jetons) n'est pas une suite d'indices de jetons disjoints de cardinal maximal. En effet, si l'on considère la configuration $x = \overleftarrow{W}\overleftarrow{W}HH\overrightarrow{W}$, $\pi_1 = \{2\}$ est bien une liste de jetons (il n'existe pas de jeton dans x disjoint de celui indexé par π_1). D'autre part, $\pi_2 = \{1, 3\}$ est également une liste de jetons, qui possède un cardinal strictement plus grand.

Dans la suite de ce chapitre, toute configuration non légitime sera dotée d'une liste de jetons π. La liste de jetons π_0 de la configuration initiale x_0 est arbitraire. Par contre, étant donnés une configuration x, une liste de jetons π de x et la réécriture de x en x' par une règle de $\mathcal{S}'$, la liste de jetons π' associée à x' est construite à partir de π comme suit :

- si l'étape de réécriture s'applique à un $H \in Jeton(\pi)$ par la règle probabiliste `R0`, on applique la stratégie de réécriture suivante :
  - si H est la première lettre d'un jeton de π, alors H est changé en $\overleftarrow{W}$,
  - si H est la deuxième lettre d'un jeton de π, alors H est changé en $\overrightarrow{W}$,

  les jetons de π sont ainsi préservés et on pose $\pi' = \pi$,
- si l'étape de réécriture crée un nouveau jeton d'indice k qui est disjoint de tout jeton de π, alors on pose $\pi' = \pi \cup \{k\}$,
- dans tous les autres cas on laisse $\pi' = \pi$.

On remarque qu'avec cette construction π' est bien une liste de jetons.

On peut définir à présent Δ_1, Δ_2 et Δ_3 comme suit :

$\underline{\Delta_1}$ **:** On pose $\Delta_1(x, \pi)$ comme N moins le nombre d'éléments de π.
Comme on va vouloir montrer que Δ_1 va diminuer et que le nombre de jetons va augmenter, il faut bien soustraire le nombre de jetons. N fait ici office de constante pour assurer que $\Delta_1(x, \pi)$ sera toujours positif.

Afin de pouvoir définir la fonction Δ_2, nous avons besoin d'introduire les notions de coefficient et de poids d'un jeton. Le *coefficient de jeton* est 4 pour D, 3 pour H, 2 pour W et 1 pour S. Ce coefficient compte le nombre d'étapes de réécriture nécessaires pour changer cette lettre en E. Le *poids d'un jeton* $\overleftarrow{\alpha}\overrightarrow{\beta}$ est la somme des coefficients de jeton de $\overleftarrow{\alpha}$ et $\overrightarrow{\beta}$: par exemple le poids du jeton $H\overrightarrow{W}$ est 5.

$\underline{\Delta_2}$ **:** On définit $\Delta_2(x, \pi)$ comme étant la somme des poids de tous les jetons de x indexés par π.

$\underline{\Delta_3}$ **:** La composante $\Delta_3(x)$ compte quant à elle le nombre de facteurs de x à deux lettres de la forme $\overleftarrow{D}H$, $\overleftarrow{D}\overleftarrow{W}$, $H\overrightarrow{D}$ ou $\overrightarrow{W}\overrightarrow{D}$.

Exemple 5.7. Dans la configuration $\overleftarrow{W}\overrightarrow{W}\overleftarrow{S}\overleftarrow{W}H\overrightarrow{D}$ de l'exemple 5.6, nous avons $\Delta_1 = 4$, $\Delta_2 = 9$ pour $\pi_1 = \{0, 3\}$, et $\Delta_1 = 4$, $\Delta_2 = 11$ pour $\pi_2 = \{0, 4\}$. Comme Δ_3 ne dépend pas de π, il a la même valeur dans les deux cas. Ici $\Delta_3 = 1$. ♣

5.6.4 Anti-jetons et composantes Δ_4, Δ_5, Δ_6

Étant donnée une configuration x, un *anti-jeton* est un facteur à deux lettres de x de la forme $\overrightarrow{\gamma}\overleftarrow{\delta} \in \overrightarrow{\Sigma'}\overleftarrow{\Sigma'}$. L'*indice* d'un anti-jeton est la position de sa première lettre $\overrightarrow{\gamma}$.

Étant donnée une liste de jetons π, on dit qu'un anti-jeton est π*-disjoint* s'il ne se chevauche avec aucun jeton de π. Dans le cas contraire il est dit *enchevêtré*.

Considérons deux jetons $\overleftarrow{\alpha}\overrightarrow{\beta}$ et $\overleftarrow{\alpha'}\overrightarrow{\beta'}$ indexés par des indices consécutifs i et i' de π. On va s'intéresser à ce qu'il se passe entre ces deux jetons (inclus) :

- soit $\overleftarrow{\alpha}\overrightarrow{\beta}$ et $\overleftarrow{\alpha'}\overrightarrow{\beta'}$ sont contigus ($i' = i + 2$) et il y a un anti-jeton $\overrightarrow{\beta}\overleftarrow{\alpha'}$ qui est dit *totalement enchevêtré* : il se chevauche avec les deux jetons,
- soit $\overleftarrow{\alpha}\overrightarrow{\beta}$ et $\overleftarrow{\alpha'}\overrightarrow{\beta'}$ ne sont pas contigus.

Dans le deuxième cas, il est facile de voir que, entre $\overleftarrow{\alpha}\overrightarrow{\beta}$ et $\overleftarrow{\alpha'}\overrightarrow{\beta'}$, il n'y a pas de facteur de la forme $\cdots\overleftarrow{\gamma}\cdots\overrightarrow{\delta}\cdots$ avec $\overleftarrow{\gamma} \in \overleftarrow{\Sigma'}$ et $\overrightarrow{\delta} \in \overrightarrow{\Sigma'}$. Sinon, il y aurait un jeton disjoint entre $\overleftarrow{\alpha}\overrightarrow{\beta}$ et $\overleftarrow{\alpha'}\overrightarrow{\beta'}$ et π ne serait pas maximal. En particulier, il ne peut pas y avoir deux H inclus strictement entre $\overleftarrow{\alpha}\overrightarrow{\beta}$ et $\overleftarrow{\alpha'}\overrightarrow{\beta'}$.

Le facteur entre $\overleftarrow{\alpha}\overrightarrow{\beta}$ et $\overleftarrow{\alpha'}\overrightarrow{\beta'}$ est donc de la forme $\overrightarrow{\Sigma'}^*\overleftarrow{\Sigma'}^*$ avec soit aucun H (cas H_0), soit un seul H (cas H_1). Plus précisément, le facteur délimité par les deux jetons est de la forme

- H_0 : $\overleftarrow{\alpha}\overrightarrow{\beta}\overrightarrow{I}\overleftarrow{I}\overleftarrow{\alpha'}\overrightarrow{\beta'}$, ou
- H_1 : $\overleftarrow{\alpha}\overrightarrow{\beta}\overrightarrow{I}H\overleftarrow{I}\overleftarrow{\alpha'}\overrightarrow{\beta'}$,

avec $\overrightarrow{I} \in \{\overrightarrow{W}, \overrightarrow{S}, \overrightarrow{D}\}^*$ et $\overleftarrow{I} \in \{\overleftarrow{W}, \overleftarrow{S}, \overleftarrow{D}\}^*$. Soient $\overrightarrow{\lambda}$ et $\overleftarrow{\mu}$ la dernière lettre de $\overrightarrow{\beta}\overrightarrow{I}$ et la première lettre de $\overleftarrow{I}\overleftarrow{\alpha'}$ respectivement. Entre les deux jetons (inclus) il y a :

- H_0 : un seul anti-jeton du type $\overrightarrow{\lambda}\overleftarrow{\mu}$ ou
- H_1 : deux anti-jetons qui se chevauchent, du type $\overrightarrow{\lambda}H$ et $H\overleftarrow{\mu}$.

On peut remarquer que le cas de l'anti-jeton totalement enchevêtré entre dans la situation H_0 : c'est le cas où $\overrightarrow{I}\overleftarrow{I}$ est le mot vide, et il n'y a bien qu'un anti-jeton dans ce cas, c'est $\overrightarrow{\beta}\overleftarrow{\alpha'}$.

Il existe un cas non répertorié ci-dessus : le cas où l'on a un jeton dans π de la forme HH. Comme $H \in \overrightarrow{\Sigma'} \cap \overleftarrow{\Sigma'}$, il se trouve que celui-ci est également un anti-jeton que l'on appelle *dégénéré*. Cependant, comme il est déjà répertorié dans la liste des jetons, il n'est pas utile de le prendre en compte dans les différentes mesures. C'est pourquoi dans la suite on ne va considérer que des anti-jetons non dégénérés.

Étant donnée une configuration x et une liste de jetons π de x, on va construire une *liste d'anti-jetons* ψ comme un ensemble d'indices obtenu en mettant, pour chaque couple de jetons consécutifs de π,

- l'indice de $\overrightarrow{\lambda}\overleftarrow{\mu}$ dans le cas H_0, et
- soit l'indice de $\overrightarrow{\lambda}H$ soit celui de $H\overleftarrow{\mu}$ dans le cas H_1.

Étant données une configuration x et une liste de jetons π de x, une liste d'anti-jetons ψ de x est donc une suite maximale d'indices d'anti-jetons disjoints et non dégénérés.

Remarque : Dans ce cas, la définition coïncide avec celle d'ensemble d'indices d'anti-jetons disjoints et non dégénérés de cardinal maximal, car on ne considère pas les anti-jetons dégénérés, et dans ce cas il y a toujours exactement un jeton indexé par ψ entre chaque couple de jetons consécutifs indexés par π.

Notons que, dans tous les cas, s'il y a un H strictement entre $\overleftarrow{\alpha}\overrightarrow{\beta}$ et $\overleftarrow{\alpha}'\overrightarrow{\beta}'$, il est compris dans un anti-jeton indexé par un élément de ψ. Formellement :

Proposition 5.8. *Pour toute configuration $x \notin \mathcal{L}'$, toute liste de jetons π et liste d'anti-jetons ψ associées, tout H de x appartient soit à un jeton de π, soit à un anti-jeton de ψ.*

Démonstration. Si H n'appartient pas à un jeton de π, il est entre deux jetons J_1 et J_2 et on est dans le cas H_1 décrit précédemment. Or dans ce cas il n'y a qu'un seul H entre J_1 et J_2 et l'anti-jeton le contient. □

Exemple 5.9. Pour la configuration $\overrightarrow{W}H\overleftarrow{S}\overrightarrow{D}\overrightarrow{W}\overleftarrow{W}\overleftarrow{D}$, il y a une liste de jetons $\pi = \{2, 6\}$ et deux listes d'anti-jetons possibles : $\psi_1 = \{0, 4\}$ et $\psi_2 = \{1, 4\}$. Les anti-jetons sont $\overrightarrow{W}H$ et $\overrightarrow{W}\overleftarrow{W}$ pour ψ_1, $H\overleftarrow{S}$ et $\overrightarrow{W}\overleftarrow{W}$ pour ψ_2. ♣

Dorénavant, toute configuration sera dotée non seulement d'une liste de jetons π mais également d'une liste d'anti-jetons ψ associée. La liste d'anti-jetons ψ_0 associée au couple initial (x_0, π_0) est choisie de façon arbitraire.

Étant donné un couple (x, π) et une liste d'anti-jetons ψ associée, la réécriture de x en x' en appliquant la règle probabiliste `R0` préserve π lorsqu'un jeton est réécrit, en suivant la stratégie de réécriture décrite à la section 5.6.3. La réécriture utilisant `R0` préserve également ψ, en utilisant la stratégie de réécriture suivante :

- si H est la première lettre d'un anti-jeton de ψ, alors H est changé en $\overrightarrow{W}$
- si H est la deuxième lettre d'un anti-jeton de ψ, alors H est changé en $\overleftarrow{W}$

On remarque facilement que, comme les anti-jetons dégénérés ne sont pas dans ψ, cette stratégie de réécriture est compatible avec celle des jetons : si un H appartient à la fois à un jeton de π et un anti-jeton de ψ, les deux stratégies de réécriture s'accordent pour le réécrire soit en $\overleftarrow{W}$ soit en $\overrightarrow{W}$. Par exemple, si H est dans $\overleftarrow{S}\,\mathbf{H}\,\overleftarrow{S}$, cette expression se réécrit en $\overleftarrow{S}\,\overrightarrow{\mathbf{W}}\,\overleftarrow{S}$.

D'autre part, comme une lettre H ne peut se trouver que dans un jeton de π ou un anti-jeton de ψ (voir proposition 5.8), la stratégie donnée ci-dessus et dans la section 5.6.3 est suffisante.

La réécriture de x en x' par une règle de $\mathcal{S}'$ transforme π en π' comme expliqué dans la section 5.6.3 et ψ en ψ' où :

- si π change, on choisit arbitrairement une nouvelle liste d'anti-jetons,
- si on réécrit la première lettre d'un anti-jeton de ψ du genre $\overrightarrow{D}\overleftarrow{W}$ d'indice i, et si la lettre d'indice $i-1$ appartient à $\overrightarrow{hold}$, alors on prend $\psi' = \psi \setminus \{i\} \cup \{i-1\}$,
- si on réécrit la deuxième lettre d'un anti-jeton de ψ du genre $\overrightarrow{W}\overleftarrow{D}$ d'indice i, et si la lettre d'indice $i+2$ appartient à $\overleftarrow{hold}$, alors on prend $\psi' = \psi \setminus \{i\} \cup \{i+1\}$,
- sinon, on pose $\psi' = \psi$.

Les deuxième et troisième cas correspondent à la situation où un anti-jeton orienté va se déplacer d'une place vers la gauche (resp. vers la droite).

Un anti-jeton est dit *orienté à gauche* (respectivement *orienté à droite*) s'il est de la forme $\{\overrightarrow{S}, \overrightarrow{D}\}\{\overleftarrow{W}, H\}$ (respectivement $\{\overrightarrow{W}, H\}\{\overleftarrow{S}, \overleftarrow{D}\}$). On les appelle ainsi car suivant la construction de ψ' décrite ci-dessus, un anti-jeton peut se déplacer vers la gauche (resp. droite) si et seulement s'il est orienté à gauche (resp. droite).

Étant donnée une liste de jetons π et un anti-jeton A d'indice k orienté à gauche (resp. à droite), la π-*distance* de A est $k - i$ (resp. $i - k$), où i est l'indice du plus proche jeton de π à gauche (resp. à droite) de A.

Nous pouvons à présent définir Δ_4, Δ_5 et Δ_6 comme suit :

$\underline{\Delta_4}$ **:** On définit $\Delta_4(x, \psi)$ comme étant N moins le nombre d'anti-jetons orientés de x indexés par ψ.
Comme pour Δ_1, on veut montrer que, quand les composantes précédentes sont constantes, le nombre d'anti-jetons orientés ne peut qu'augmenter (ou rester constant) d'où le fait qu'on prend l'opposé de ce nombre.

$\underline{\Delta_5}$ **:** $\Delta_5(x, \pi, \psi)$ vaut pour sa part la somme des π-distances de tous les anti-jetons orientés.

Afin de pouvoir définir la fonction Δ_6, nous avons besoin des notions de coefficient et de poids d'un anti-jeton. Le *coefficient d'anti-jeton* est 4 pour H, 3 pour W, 2 pour S et 1 pour D. Il compte le nombre de réécritures nécessaires de cette lettre avant de modifier une des mesures Δ_1 à Δ_5. Le *poids d'un anti-jeton* $\overrightarrow{\alpha}\overleftarrow{\beta}$ est la somme des coefficients d'anti-jetons de $\overrightarrow{\alpha}$ et $\overleftarrow{\beta}$, par exemple le poids de $H\overleftarrow{W}$ est 7.

$\underline{\Delta_6}$ **:** Définissons $\Delta_6(x, \psi)$ comme la somme des poids de tous les anti-jetons de x indexés par ψ.

Exemple 5.10. Dans la configuration $\overrightarrow{W}H\overleftarrow{S}\overrightarrow{D}\overrightarrow{W}\overleftarrow{W}\overleftarrow{D}$ de l'exemple 5.9, nous avons $\Delta_4 = 7(= N)$, $\Delta_5 = 0$ et $\Delta_6 = 13$ pour $\psi_1 = \{0, 4\}$, et $\Delta_4 = 6$, $\Delta_5 = 1$, $\Delta_6 = 12$ pour $\psi_2 = \{1, 4\}$. Pour ψ_1, aucun anti-jeton n'est orienté. Pour ψ_2, l'anti-jeton $H\overleftarrow{S}$ est orienté à droite. Sa π-distance avec le jeton $\overleftarrow{S}\overrightarrow{D}$ est 1. ♣

5.6.5 Fonction Δ et preuve de progrès

Pour pouvoir montrer que la fonction Δ décroît pour chaque transition, il faut une composante de Δ qui puisse varier lorsque l'on réécrit une lettre qui ne se trouve ni dans un jeton, ni dans un anti-jeton. C'est le rôle de Δ_7.

On associe à chaque lettre une valeur appelée WSD-coefficient, qui vaut 3 pour W, 2 pour S, 1 pour D et 0 pour H. Cette valeur compte le nombre de réécritures nécessaires pour créer un nouveau H, et elle est définie ainsi car la création d'un nouveau H hors des jetons et anti-jetons de π et ψ va créer un nouveau jeton, disjoint de tous ceux de π (et donc faire diminuer Δ_1).

$\underline{\Delta_7}$ **:** La dernière composante $\Delta_7(x)$ est la somme des WSD-coefficients de toutes les lettres de x.

Exemple 5.11. Dans la configuration $\overrightarrow{W}H\overleftarrow{S}\overrightarrow{D}\overrightarrow{W}\overleftarrow{W}\overleftarrow{D}$, nous avons $\Delta_7 = 13$. ♣

Nous pouvons à présent décrire la fonction Δ :

$\underline{\Delta}$ **:** Étant donnée une configuration $x \notin \mathcal{L}'$, une liste de jetons π et une liste d'anti-jetons ψ associées, la fonction Δ est définie comme un 7-uplet $(\Delta_1, \Delta_2, \Delta_3, \Delta_4, \Delta_5, \Delta_6, \Delta_7)$.

Afin de prouver que la fonction Δ décroît (pour un ordre que nous allons définir) à chaque transition en suivant notre stratégie de réécriture, nous allons utiliser le lemme suivant :

Lemme 5.12. *Soient $x \notin \mathcal{L}'$ une configuration et π une liste de jetons de x. Soient x' une configuration telle que $x \to x'$ et π' une liste de jetons de x' définie à partir de π comme décrit à la section 5.6.3. Si Δ_1 et Δ_2 restent constantes lors de cette transition, alors $\Delta_3(x') \leq \Delta_3(x)$.*

Démonstration. Par contraposée : étant donné $x \notin \mathcal{L}'$, une liste de jetons π et une configuration x' telle que $x \to x'$ et $\Delta_3(x') > \Delta_3(x)$, on va montrer que la liste de jetons π' de x', construite à partir de π comme décrit à la section 5.6.3 satisfait $\Delta_1(x', \pi') < \Delta_1(x, \pi)$ ou $\Delta_2(x', \pi') < \Delta_2(x, \pi)$.

Vu qu'on a supposé $\Delta_3(x') > \Delta_3(x)$, cela signifie qu'un motif de la forme $\{\overrightarrow{W}, H\}\overrightarrow{D}$ ou $\overleftarrow{D}\{\overleftarrow{W}, H\}$ est apparu dans x' qui n'était pas dans x. Par symétrie, on ne considère que le cas où $x' = x_1\{\overrightarrow{W}, H\}\overrightarrow{D}x_2$ où x_1 et x_2 appartiennent à Σ'^*. Cela implique que $x = x_1\lambda\mu x_2$ avec $\lambda, \mu \in \Sigma'$ tels que $\lambda\mu$ se réécrit en $\{\overrightarrow{W}, H\}\overrightarrow{D}$. On a donc :

- soit $\lambda \in \{\overrightarrow{W}, H\}$ et μ est réécrit en $\overrightarrow{D}$,
- soit λ est réécrit en $\overrightarrow{W}$ et $\mu = \overrightarrow{D}$,
- soit λ est réécrit en H et $\mu = \overrightarrow{D}$.

Le premier cas est impossible car pour réécrire μ en $\overrightarrow{D}$ il faut que $\mu = \overrightarrow{S}$ et une règle $\lambda\overrightarrow{S} \to \lambda\overrightarrow{D}$ ne peut être appliquée que si $\lambda \in \overrightarrow{hold}$, ce qui n'est pas le cas ici.
Dans le deuxième cas, la règle appliquée est `R0` et $\lambda = H$. Alors $\lambda\mu = H\overrightarrow{D}$ et $\Delta_3(x') = \Delta_3(x)$ ce qui contredit notre hypothèse sur Δ_3.
Le seul cas possible reste donc le troisième : $\lambda\mu = \lambda\overrightarrow{D}$ réécrit en $H\overrightarrow{D}$. Cela signifie que λ correspond à une lettre D, et on doit encore considérer deux cas : $\lambda \in Jeton(\pi)$ ou $\lambda \notin Jeton(\pi)$.
Si $\lambda \in Jeton(\pi)$ alors selon notre construction on prend $\pi' = \pi$ et on a $\Delta_2(x', \pi') < \Delta_2(x, \pi)$.
Si λ n'appartient pas à un jeton indexé par π, alors μ $(= \overrightarrow{D})$ non plus. Il ne peut pas être la deuxième lettre d'un jeton car λ n'en est pas la première, et du fait de son orientation il ne peut pas être la première lettre d'un jeton non plus. La lettre λ doit être orientée à droite sinon $\lambda\mu$ serait un jeton de x disjoint de tout jeton de π ce qui est impossible. On a donc $\lambda\mu = \overrightarrow{D}\overrightarrow{D}$ qui est réécrit en $H\overrightarrow{D}$. Un nouveau jeton d'indice k disjoint de tous ceux de π vient d'être créé et, suivant la construction définie à la section 5.6.3 on prend $\pi' = \pi \cup \{k\}$, d'où $\Delta_1(x', \pi') < \Delta_1(x, \pi)$.

On a donc bien montré que, dans chaque cas, si $\Delta_3(x') > \Delta_3(x)$ alors $\Delta_1(x', \pi') < \Delta_1(x, \pi)$ ou $\Delta_2(x', \pi') < \Delta_2(x, \pi)$. □

D'après la proposition 5.8, toute configuration peut être décomposée en des jetons de π, des anti-jetons de ψ, et des lettres W, S, D en dehors de tous ces jetons et anti-jetons. En utilisant cette décomposition, on va montrer, en distinguant les différents cas de réécriture possibles (suivant la lettre réécrite et le contexte), qu'avec notre stratégie de réécriture, pour tous (x, π, ψ), Δ décroît quand x se réécrit en x' et les listes π' et ψ', associées à x', sont construites à partir de π et ψ comme décrit dans les sections 5.6.3 et 5.6.4.

Formellement, on va définir un ordre, $\ll$, pour classer les valeurs de Δ. On dit que $\Delta(x', \pi', \psi') \ll \Delta(x, \pi, \psi)$ lorsque il existe $i \in \{1, .., 7\}$ tel que

$$\Delta_i(x', \pi', \psi') < \Delta_i(x, \pi, \psi) \text{ et } \forall j < i,\ \Delta_j(x', \pi', \psi') = \Delta_j(x, \pi, \psi)$$

L'ordre $\ll$ est appelé extension lexicographique de $<$. On a alors,

Proposition 5.13. *Pour tout $x \notin \mathcal{L}'$, toute liste de jetons π et toute liste d'anti-jetons ψ de x, pour toute position i dans $\mathcal{E}(x)$, il existe une configuration x', une liste de jetons π' et une liste d'anti-jetons ψ' de x' telles que :*

$$x \xrightarrow[S']{i} x' \wedge (x' \in \mathcal{L}' \ \vee\ \Delta(x', \pi', \psi') \ll \Delta(x, \pi, \psi)).$$

Démonstration. Considérons une configuration non légitime x, ainsi qu'une liste de jetons π et une liste d'anti-jetons ψ associées. Supposons que $x \xrightarrow[S']{i} x'$ en suivant le cas échéant la stratégie de réécriture décrite dans les sections 5.6.3 et 5.6.4 pour la règle probabiliste `R0`. Pour prouver la proposition on n'a besoin de considérer que le cas où x' n'est pas légitime (et donc les règles `R5` et `R7` ne sont pas utilisées), et x n'est pas légitime (la règle `R11` n'est pas utilisée non plus).
Comme annoncé précédemment (voir page 99), dans cette preuve nous allons supposer que x ne contient pas de L_1 ou L_2 (et donc les règles `R12` et `R13'` ne sont pas utilisées). Les modifications pour prendre en compte L_1 et L_2 sont données à la fin de la preuve, page 107.

Nous allons montrer que, pour toute règle de la forme $S \to D$ (i.e. : `R6`, `R8`), $D \to H$ (i.e. : `R9`, `R10`), $H \to W$ (i.e. : `R0`) ou $W \to S$ (i.e. : `R1`, `R3`), on a $\Delta(x', \pi', \psi') \ll \Delta(x, \pi, \psi)$.
La preuve se fait par disjonction de cas, selon la position de la lettre changée par la transition : dans un jeton, dans un anti-jeton (mais pas un jeton) ou partout ailleurs.

Dans la suite, lorsque l'on dira qu'une composante Δ_i $(2 \leq i \leq 7)$ de Δ décroît, il sera sous-entendu que les composantes précédentes (Δ_j pour $j < i$) restent constantes. Quand les changements de π (resp. ψ) ne seront pas précisés, on supposera que $\pi' = \pi$ (resp. $\psi' = \psi$).
Considérons les différents endroit où la réécriture peut avoir lieu :

1. Dans un jeton $\overleftarrow{\alpha}\overrightarrow{\beta}$ de π. Par symétrie, on ne considère que le cas où la lettre réécrite est la première du jeton.
   - la réécriture par la règle probabiliste $H \to \overleftarrow{W}$ (`R0`) préserve Δ_1 grâce à notre stratégie de réécriture, et fait décroître Δ_2,
   - la réécriture par la règle $\overleftarrow{D} \to H$ ou $\overleftarrow{W} \to \overleftarrow{S}$ diminue Δ_2,
   - la réécriture de $\overleftarrow{\alpha}$ *via* `R6` : $\overleftarrow{S} \to \overleftarrow{D}$ est impossible car, comme $\overrightarrow{\beta} \in \overrightarrow{\Sigma'}$, la baguette à la droite de $\overleftarrow{\alpha}$ est libre, et donc si on réécrivait $\overleftarrow{S}$, ce serait en E (et on arriverait dans $\mathcal{L}$).

2. Dans un anti-jeton A de ψ d'indice i.

   (a) Si A est un anti-jeton π-disjoint $\overrightarrow{\gamma}\overleftarrow{\delta}$. Comme on ne prend que les configurations acceptables, on a $\overrightarrow{\gamma} \in \{\overrightarrow{W}, \overrightarrow{S}, \overrightarrow{D}\}$ et $\overleftarrow{\delta} \in \{H, \overleftarrow{W}\}$, ou $\overrightarrow{\gamma} \in \{H, \overrightarrow{W}\}$ et $\overleftarrow{\delta} \in \{\overleftarrow{W}, \overleftarrow{S}, \overleftarrow{D}\}$). À part si on crée un nouveau jeton (ce qui ferait décroître Δ_1), comme on ne touche pas aux jetons de π, on a Δ_1 et Δ_2 qui restent constantes et, d'après le lemme 5.12, Δ_3 ne peut pas augmenter. Par symétrie, nous n'allons considérer que les cas où la lettre réécrite est la première de l'anti-jeton.
   - $A \in H\{\overleftarrow{W}, H\} \to \overrightarrow{W}\{\overleftarrow{W}, H\}$. En suivant notre stratégie de réécriture, c'est la seule orientation possible pour W. L'anti-jeton reste non orienté et Δ_6 diminue.
   - $A = H\overleftarrow{hold} \to \overrightarrow{W}\overleftarrow{hold}$. En suivant notre stratégie de réécriture, c'est la seule orientation possible pour W. Comme le jeton était orienté il le reste et garde la même π-distance, donc Δ_4 et Δ_5 restent constantes, par contre Δ_6 diminue.
   - $A \in \overrightarrow{W}\{\overleftarrow{W}, H\} \to \overrightarrow{S}\{\overleftarrow{W}, H\}$. Dans ce cas l'anti-jeton devient orienté (à gauche) et Δ_4 diminue.
   - $A = \overrightarrow{W}\overleftarrow{hold}$, il est impossible de réécrire le $\overrightarrow{W}$ car la baguette concernée est prise.
   - $A \in \overrightarrow{S}\{\overleftarrow{W}, H\} \to \overrightarrow{D}\{\overleftarrow{W}, H\}$ l'anti-jeton garde son orientation et sa π-distance, mais Δ_6 diminue.
   - $A = \overrightarrow{D}\overleftarrow{W} \to H\overleftarrow{W}$, on a un jeton orienté à gauche, et on a besoin de savoir quelle est la lettre λ à gauche de A. Tout d'abord $\lambda \in \overrightarrow{\Sigma'}$ sinon on aurait un motif dans

$\{\overleftarrow{W}, \overleftarrow{S}, \overleftarrow{D}\}\overrightarrow{D}$ qui serait un jeton de x disjoint de ceux de π.

- Si $\lambda \in \overrightarrow{hold}$, on prend $\psi' = \psi \setminus \{i\} \cup \{i-1\}$, notre nouvel anti-jeton est λH, également orienté à gauche mais Δ_5 décroît de 1.
- Si $\lambda = \overrightarrow{W}$, Δ_3 diminue
- Si $\lambda = H$ alors soit on crée un nouveau jeton disjoint de ceux de π et on prend $\pi' = \pi \cup \{i-1\}$, soit (si λ appartient à un jeton de π) Δ_3 diminue.

– $A = \overrightarrow{D}H \to HH$, comme A est π-disjoint, on a créé un nouveau jeton disjoint de ceux de π. On prend donc $\pi' = \pi \cup \{i\}$, ψ' est une liste d'anti-jetons arbitraire associée à π, et Δ_1 décroît.

(b) Si A se chevauche avec au moins un jeton de π, on n'a besoin de considérer la réécriture que de la lettre qui n'est pas dans $Jeton(\pi)$ (le cas d'une lettre de $Jeton(\pi)$ a été traité au cas 1). En particulier, si l'anti-jeton est totalement enchevêtré, il n'y a rien à faire.

Toujours pour des arguments de symétrie, on ne va considérer que le cas où $A = \overrightarrow{\gamma}\overleftarrow{\alpha}$, tel que $\overleftarrow{\alpha}$ fait partie d'un jeton, et $\overrightarrow{\gamma}$ non. Pour les mêmes raisons que dans le cas 2(a), tant que l'on ne va pas changer π, Δ_1 et Δ_2 restent constantes.

– Si $\overrightarrow{\gamma} = H$, avec notre stratégie de réécriture, il se réécrit en $\overrightarrow{W}$. Ainsi $A = H\overleftarrow{\alpha}$ devient $A' = \overrightarrow{W}\overleftarrow{\alpha}$. Cette réécriture ne change rien à l'éventuelle (non-)orientation ni à la π-distance. On a donc Δ_4 et Δ_5 qui restent constantes pendant que Δ_6 diminue.

– Si $\overrightarrow{\gamma} = \overrightarrow{W}$, la réécriture n'est possible que si $\overleftarrow{\alpha} \in \{H, \overleftarrow{W}\}$. Ainsi $A = \overrightarrow{W}\overleftarrow{\alpha}$ devient $A' = \overrightarrow{S}\overleftarrow{\alpha}$. Contrairement à A, A' est orienté, d'où Δ_4 diminue.

– Si $\overrightarrow{\gamma} = \overrightarrow{S}$, l'anti-jeton devient $\overrightarrow{D}\overleftarrow{\alpha}$, conserve son orientation, et Δ_6 décroît.

– Si $\overrightarrow{\gamma} = \overrightarrow{D}$, le cas est similaire à la réécriture $\overrightarrow{D} \to H$ étudiée dans le cas d'un anti-jeton π-disjoint (cas 2(a)) :

- Si $\lambda \in \overrightarrow{hold}$, on prend $\psi' = \psi \setminus \{i\} \cup \{i-1\}$, notre nouvel anti-jeton est λH, également orienté à gauche mais Δ_5 décroît de 1.
- Si $\lambda = \overrightarrow{W}$, Δ_3 diminue
- Si $\lambda = H$ alors soit on crée un nouveau jeton disjoint de ceux de π et on prend $\pi' = \pi \cup \{i-1\}$ et ψ' arbitraire associé à π', soit (lorsque λ appartient à un jeton de π) Δ_3 diminue.

3. Hors des jetons et anti-jetons :

Dans ce cas soit un nouveau jeton disjoint de ceux de π est créé, soit $\Delta_1, \Delta_2, \Delta_4, \Delta_5$ et Δ_6 restent constants. Plus précisément, soit λ la lettre modifiée et i sa position. D'après la proposition 5.8, λ appartient à l'ensemble $\{W, S, D\}$. On considère donc les trois cas suivants :

– $\lambda = \overrightarrow{W}$: la réécriture de $\overrightarrow{W}$ en $\overrightarrow{S}$ fait diminuer soit Δ_3 si la lettre à droite de λ est $\overrightarrow{D}$, soit Δ_7.

– $\lambda = \overrightarrow{S}$: la réécriture de $\overrightarrow{S}$ en $\overrightarrow{D}$ fait diminuer Δ_7.

– $\lambda = \overrightarrow{D}$: étant donné que λ n'appartient ni à un jeton de π ni à un anti-jeton de ψ, la lettre μ à sa droite appartient à $\overrightarrow{\Sigma'}$ (sinon $\lambda\mu$ serait un anti-jeton (de ψ car non dégénéré et ne contenant pas de H)). La réécriture de $\overrightarrow{D}$ en H donne donc $H\mu \in \overleftarrow{\Sigma'}\overrightarrow{\Sigma'}$. Cela donne un nouveau jeton disjoint de ceux de π et d'indice i. On pose alors $\pi' = \pi \cup \{i\}$, on prend pour ψ' une liste d'anti-jetons arbitraire définie à partir de π' et on a Δ_1 qui diminue.

– Les cas de $\lambda = \overleftarrow{W}$, $\overleftarrow{S}$ ou $\overleftarrow{D}$ se ramènent par symétrie aux cas précédemment traités.

Ceci termine la preuve sous l'hypothèse que la configuration x ne contient pas de lettre L_1 ou L_2. Cette preuve peut facilement être étendue pour prendre en compte les lettres L_1 et L_2 comme on va le montrer dans ce qui suit.

La lettre L_2, correspondant à un philosophe tenant encore la baguette de droite mais sur le point de la reposer, va informellement être "associée" à $\overrightarrow{D}$. En particulier elle va apparaître dans les mêmes motifs, et recevoir le même coefficient dans un jeton, un anti-jeton ou ailleurs. La lettre L_1, du fait qu'elle correspond à un philosophe tenant ses deux baguettes, va représenter un motif $\leftarrow\rightarrow$ (donc un jeton) à elle toute seule. Les définitions sont donc changées de la manière suivante :

- La lettre L_2 est ajouté à $\overrightarrow{\Sigma'}$.
- Un jeton est maintenant soit facteur appartenant à $\overleftarrow{\Sigma'}\overrightarrow{\Sigma'}$, soit L_1.
- Pour les anti-jetons, on va considérer des facteurs (acceptables) $\overrightarrow{\gamma}\overleftarrow{\delta}$ avec $\overrightarrow{\gamma} \in \overrightarrow{\Sigma'} \cup \{L_1\}$ (car maintenant L_2 est dans $\overrightarrow{\Sigma'}$) et $\overleftarrow{\delta} \in \overleftarrow{\Sigma'} \cup \{L_1\}$.
- La définition d'anti-jeton orienté inclut maintenant des facteurs du type $\{L_1, L_2\}\{\overleftarrow{W}, H\}$ (orienté à gauche) et $\{\overrightarrow{W}, H\}L_1$ (orienté à droite).

La définition de π-distance reste la même si l'on considère la nouvelle définition de jeton et d'anti-jeton orienté. De même, pour toute liste de jetons π et toute liste d'anti-jetons associée ψ, avec les nouvelles définitions on a toujours exactement un anti-jeton de ψ entre deux jetons de π.

La fonction Δ est changée en $\Delta' = (\Delta'_1, \Delta'_2, \Delta'_3, \Delta'_4, \Delta'_5, \Delta'_6, \Delta'_7)$, avec :

- $\Delta'_1(x, \pi)$ est définie comme $N -$ (nb de jetons classiques) + (nb de L_1). Le nombre de processus dans l'était L_1 est ajouté (et non soustrait) car, contrairement aux jetons classiques, les L_1 ne vont que disparaître quand on n'est pas dans $\mathcal{L}'$,
- $\Delta'_2(x, \pi)$ est définie comme $\Delta_2(x, \pi)$ sauf que l'on ajoute les coefficients de jetons suivants : 0 pour L_1 et 4 (comme pour D) pour L_2,
- $\Delta'_3(x)$ compte, en plus de $\Delta_3(x)$, les motifs de la forme HL_2 ou $\overrightarrow{W}L_2$ de x,
- $\Delta'_4(x, \psi)$ est définie comme $\Delta_4(x, \psi)$, seule la définition d'anti-jetons orientés a changé,
- $\Delta'_5(x, \pi, \psi)$ est définie comme $\Delta_5(x, \pi, \psi)$, seule la définition des jetons et anti-jetons pour calculer la π-distance a changé,
- $\Delta'_6(x, \psi)$ compte toujours la somme des poids des anti-jetons de x indexés par ψ. Le coefficient d'anti-jeton est 1 pour L_2 et 0 pour L_1.
- $\Delta'_7(x)$ vaut toujours la somme des WSD-coefficients de toutes les lettres de x. Le WSD-coefficient est 1 pour L_2 et 0 pour L_1.

On remarque que tous les coefficients associés à L_1 sont 0 quel que soit l'endroit où on le considère. En fait on pourrait leur donner la valeur que l'on veut, car de toute façon, quand un L_1 est réécrit, Δ_1 décroît et par conséquent l'évolution des autres composantes n'est pas pertinente. Comme annoncé, dans tous les cas L_2 a le même coefficient que D. □

Le théorème 5.3 à propos de la propriété de progrès se déduit alors de la proposition 5.13 et du théorème 5.5.

5.7 Temps moyen de convergence

Dans les approches traditionnelles du dîner des philosophes probabilistes (voir par exemple [Lyn96]), le temps est mesuré en terme de tours (intervalles pendant lesquels chaque processus est sélectionné au moins une fois). Le temps n'est jamais évalué en nombre de transitions. Lynch, Saias et Segala ont donné dans [LSS94] une preuve montrant que 63 est une borne supérieure du nombre moyen de tours nécessaires pour atteindre l'ensemble $\mathcal{L}$. Dans l'article [PS95], cette méthode de preuve est améliorée pour permettre de vérifier certaines parties automatiquement.

Dans notre approche, nous ne supposons plus l'existence de tels tours, et nous cherchons à calculer le temps moyen de convergence en nombre de transitions. Or il se trouve que pour un certain ordonnancement "malveillant", ce temps de convergence peut être "très" long.

Nous allons montrer sur un exemple que le temps moyen de convergence est au moins exponentiel en N (qui est le nombre de philosophes dans le système) pour un certain ordonnancement "malveillant". Nous allons exhiber un ordonnancement qui, partant de la configuration initiale $x_0 = \overrightarrow{S}^{N-2}\overleftarrow{W}\overleftarrow{S}$, va rester dans l'ensemble de configurations $\{\overrightarrow{S}^i\overleftarrow{W}\overleftarrow{S}^j | i+j+1 = N\} \cup \{\overrightarrow{S}^i\overrightarrow{W}\overleftarrow{S}^j | i+j+1 = N\}$ (et donc hors de $\mathcal{L}$) pendant un temps moyen exponentiel en N.

Étudions deux cas pouvant se produire à partir d'une configuration de la forme $\overrightarrow{S}^i\overleftarrow{W}\overleftarrow{S}^j$ avec $i \geq 2$ et $i+j+1 = N$. Ces deux cas consistent en l'application de 4 règles consécutives, l'une d'entre elles étant probabiliste. L'ordonnancement sélectionne d'abord la position du $\overrightarrow{S}$ le plus à droite, et ce trois fois de suite. Comme il y a encore un $\overrightarrow{S}$ à la gauche de la position réécrite, les règles appliquées à cette même position sont successivement `R8` : $\overrightarrow{S}\overrightarrow{S} \rightarrow \overrightarrow{S}\overrightarrow{D}$, `R10` : $\overrightarrow{D} \rightarrow H$ et `R0` : $H \rightarrow (\overleftarrow{W}$ ou $\overrightarrow{W})$. Cela donne deux configurations possibles, $\overrightarrow{S}^{i-1}\overleftarrow{W}\overleftarrow{W}\overleftarrow{S}^j$ ou $\overrightarrow{S}^{i-1}\overrightarrow{W}\overleftarrow{W}\overleftarrow{S}^j$, suivant le résultat de la règle probabiliste `R0`.
À partir de $\overrightarrow{S}^{i-1}\overleftarrow{W}\overleftarrow{W}\overleftarrow{S}^j$, l'ordonnancement sélectionne le $\overleftarrow{W}$ de droite ce qui donne (par application de `R1`) $\overrightarrow{S}^{i-1}\overleftarrow{W}\overleftarrow{S}^{j+1}$. À partir de $\overrightarrow{S}^{i-1}\overrightarrow{W}\overleftarrow{W}\overleftarrow{S}^j$, l'ordonnancement va, selon les valeurs respectives de i et j choisir $\overrightarrow{W}$ ou $\overleftarrow{W}$, ce qui engendre les deux cas suivants[2] :

(A) l'ordonnancement sélectionne la position de $\overleftarrow{W}$, où l'on applique `R1`, ce qui donne $\overrightarrow{S}^{i-1}\overrightarrow{W}\overleftarrow{S}^{j+1}$, ou bien

(B) l'ordonnancement sélectionne la position de $\overrightarrow{W}$, où l'on applique `R3`, ce qui donne $\overrightarrow{S}^i\overleftarrow{W}\overleftarrow{S}^j$.

De manière symétrique, à partir d'une configuration du type $\overrightarrow{S}^i\overrightarrow{W}\overleftarrow{S}^j$ (avec $j \geq 2$ et $i+j+1 = N$), en appliquant 3 règles consécutives à la position du $\overleftarrow{S}$ le plus à gauche, les règles en question sont `R6`, `R9` et `R0` et l'on arrive dans $\overrightarrow{S}^i\overrightarrow{W}\overleftarrow{W}\overleftarrow{S}^{j-1}$ ou $\overrightarrow{S}^i\overrightarrow{W}\overrightarrow{W}\overleftarrow{S}^{j-1}$.

À partir de $\overrightarrow{S}^i\overrightarrow{W}\overrightarrow{W}\overleftarrow{S}^{j-1}$, l'ordonnancement sélectionne le $\overrightarrow{W}$ le plus à gauche, la règle appliquée est `R3` et on obtient $\overrightarrow{S}^{i+1}\overrightarrow{W}\overleftarrow{S}^{j-1}$. À partir de $\overrightarrow{S}^i\overrightarrow{W}\overleftarrow{W}\overleftarrow{S}^{j-1}$, suivant les valeurs de i et j l'ordonnancement va choisir entre les deux cas suivants :

(A') soit il sélectionne $\overrightarrow{W}$ auquel est appliqué la règle `R3` et on arrive dans $\overrightarrow{S}^{i+1}\overleftarrow{W}\overleftarrow{S}^{j-1}$,

(B') soit il sélectionne $\overleftarrow{W}$ auquel est appliqué la règle `R1` et on arrive dans $\overrightarrow{S}^i\overrightarrow{W}\overleftarrow{S}^j$.

Le démon peut permettre d'itérer un tel comportement tant que le système n'atteint pas une configuration du type :

---

[2] Le choix que va faire l'ordonnancement est entièrement déterminé par les va leurs de i et j, comme on va le voir par la suite. On a donc bien toujours un ordonnancement déterministe.

$x_{end} = \overrightarrow{S}^{N-2}\overrightarrow{W}\overleftarrow{S}$ ou $y_{end} = \overrightarrow{S}\overleftarrow{W}\overleftarrow{S}^{N-2}$.

Considérons l'ordonnancement “malveillant” qui tente de se maintenir dans des configurations du type $\{\overrightarrow{S}^i\overleftarrow{W}\overleftarrow{S}^j | i+j+1=N\} \cup \{\overrightarrow{S}^i\overrightarrow{W}\overleftarrow{S}^j | i+j+1=N\}$ le plus longtemps possible. À partir d'une configuration $\overrightarrow{S}^i\overleftarrow{W}\overleftarrow{S}^j$ (avec $i \geq 2$ et $i+j+1=N$), il choisit entre (A) et (B) selon les valeurs respectives de i et j, plus précisément :

- il choisit (A) si $j \geq i$,
- il choisit (B) si $j < i$.

De manière symétrique, à partir de $\overrightarrow{S}^i\overrightarrow{W}\overleftarrow{S}^j$ (avec $j \geq 2$ et $i+j+1=N$) :

- il choisit (A') si $i \geq j$,
- il choisit (B') si $i < j$.

Pour cet ordonnancement, calculons le temps moyen pour atteindre une configuration x_{end} ou y_{end}. Partant d'une configuration $\overrightarrow{S}^i\overrightarrow{W}\overleftarrow{S}^j$ (resp. $\overrightarrow{S}^i\overleftarrow{W}\overleftarrow{S}^j$), avec $j \geq 2$ (resp. $i \geq 2$) et $i+j+1=N$, on a :

$E[\overrightarrow{i}] = 4 + 1/2\ E[\overrightarrow{i+1}] + 1/2\ E[\overleftarrow{i+1}]$, pour $1 \leq i < N-2$ et $2i \geq N-1$.
$E[\overrightarrow{i}] = 4 + 1/2\ E[\overrightarrow{i+1}] + 1/2\ E[\overrightarrow{i}]$, pour $1 \leq i < N-2$ et $2i < N-1$.
$E[\overleftarrow{i}] = 4 + 1/2\ E[\overrightarrow{i-1}] + 1/2\ E[\overleftarrow{i-1}]$, pour $2 \leq i$ et $2i \leq N-1$.
$E[\overleftarrow{i}] = 4 + 1/2\ E[\overleftarrow{i-1}] + 1/2\ E[\overleftarrow{i}]$, pour $2 \leq i$ et $2i > N-1$.
$E[\overrightarrow{N-2}] = 0$.
$E[\overleftarrow{1}] = 0$.

On résout alors ce système linéaire. La symétrie implique que $E[\overrightarrow{i}] = E[\overleftarrow{N-1-i}]$, pour tout $1 \leq i < N-1$. Soit m la partie entière de $N/2-1$, le résultat est :

$E[\overrightarrow{i}] = 8(2^{N-3-m}(2m-N+6) - i - 2)$, pour $1 \leq i \leq m$.
$E[\overrightarrow{i}] = 8((2m-N+6)(2^{N-3-m} - 2^{i-m-1}) - N + i + 2)$ pour $m+1 \leq i < N-1$.

En particulier, le temps moyen pour aller de x_0 à x_{end} est :

$$E[\overleftarrow{N-2}] = E[\overrightarrow{1}] = 8(2^{N-3-m}(2m-N+6) - 3) \geq 2^{N/2}.$$

Ceci implique que, pour cet ordonnancement, on peut rester hors de $\mathcal{L}'$ un nombre moyen de transitions exponentiel, et donc la borne supérieure du temps moyen de convergence pour un ordonnancement arbitraire est au moins exponentiel.

Conclusion

Ce résultat confirme donc l'idée que nous nous faisions sur le manque de précision de l'évaluation du temps moyen en “tours”. On peut en effet imaginer un ordonnancement pour l'algorithme original qui suive l'ordre d'application des règles ci-dessus sauf que, toutes les 2^N étapes, il sélectionne chacun des processus tour à tour. Cet ordonnancement aurait donc un temps moyen de convergence constant si on l'évalue en nombre de tours, mais de l'ordre de l'exponentielle si on l'évalue en nombre de transitions. Il existe donc des ordonnancements, même équitables, qui font que le système prend un temps moyen très important avant de converger.

5.8 Le dîner des philosophes courtois

Dans l'article [LR81], Lehmann et Rabin présentent un deuxième algorithme probabiliste, l'algorithme des "*philosophes courtois*", assurant une propriété différente (qu'ils appellent "lockout freedom") : si un philosophe à faim, alors pour tout ordonnancement équitable, avec probabilité 1, ce philosophe mangera un jour.

Cette propriété est plus forte que celle prouvée dans le précédent algorithme, et elle se base sur le fait qu'à chaque baguette correspondent maintenant deux variables partagées : une caractérisant le fait que la baguette est prise ou non, et l'autre désignant parmi les deux philosophes lequel a mangé en dernier.

Avant de prendre sa première baguette, un philosophe va alors regarder si celle-ci est libre puis, si elle est libre regarder si son voisin a faim. Si ce voisin n'a pas faim, le philosophe prend la baguette. Si le voisin a également faim, le philosophe va regarder l'état de la deuxième variable partagée de la baguette. Si elle indique que c'est le voisin qui a mangé en dernier, le philosophe prend la baguette, sinon il attend que son voisin ait mangé. De plus, au moment de reposer les baguettes, un philosophe qui vient de manger met les variables partagées à jour, signifiant qu'il a mangé en dernier.

Remarque : Ce nouvel algorithme, du fait de la notion de "courtoisie", nécessite pour chaque philosophe une plus grande connaissance sur l'état de ses voisins. Outre les deux variables associées à chacune de ses deux fourchettes partagées, chaque philosophe doit aussi savoir l'état de ses voisins. En effet, lorsqu'un philosophe veut prendre une première fourchette et que la variable lui dit qu'il est le dernier à avoir mangé, il doit regarder si son voisin a faim ou non, et en fonction de cela, laisser la fourchette ou la prendre.

Cet algorithme semble assez similaire au précédent. Cependant, même en enlevant les transitions invariantes, il ne vérifie pas la nouvelle condition de progrès si on ne suppose pas l'ordonnancement équitable. En effet, tout d'abord on ne peut pas ici fusionner H et T car ils ne jouent pas le même rôle : on doit parfois céder son tour à un philosophe qui a faim, mais pas à un qui pense. Considérons l'exécution suivante, similaire à celle de l'exemple 5.2, mais où l'on n'a pas supprimé les H
$T\overleftarrow{W}TTT \to T\overleftarrow{W}THT \to T\overleftarrow{W}T\overleftarrow{W}T \to T\overleftarrow{W}T\overleftarrow{S}T \to T\overleftarrow{W}TET \to T\overleftarrow{W}TL_1T \to T\overleftarrow{W}TL_2T \to T\overleftarrow{W}TTT \to T\overleftarrow{W}THT \to T\overleftarrow{W}T\overleftarrow{W}T \to T\overleftarrow{W}T\overleftarrow{S}T \to T\overleftarrow{W}TET \to T\overleftarrow{W}TL_1T$...
elle suit parfaitement l'algorithme des philosophes courtois (les voisins du philosophe numéro 4 restent à penser et donc le numéro 4 peut manger tant qu'il le souhaite). Par contre le philosophe numéro 2 n'est jamais choisi alors qu'il a faim. Il ne va donc jamais manger, et la nouvelle condition de progrès n'est pas vérifiée.

Ceci montre que, lorsque l'on veut qu'une propriété de progrès soit vraie individuellement pour chaque philosophe, il faut soit que l'on suppose l'ordonnancement équitable, soit que le système ne permette que des ordonnancements équitables (comme dans le cas de l'algorithme de [BDLGJ02]). Ici il faudrait que l'ordonnancement sélectionne infiniment souvent chaque philosophe qui a faim.

Cet algorithme du dîner des philosophes courtois se prête par contre à une vérification en utilisant une méthode d'abstraction appelée invariant de réseau, comme étudié dans [ZPK02, KPSZ02, APZ03].

Chapitre 6

Intérêt des algorithmes sans équité

L'étude d'une variante de l'algorithme des "free philosophers" de Lehmann et Rabin nous adonné l'occasion de nous pencher sur le cas des algorithmes sans équité. À ce sujet, on peut raisonnablement se poser la question suivante :

Pourquoi s'intéresser à des algorithmes sans équité ?

En effet, dans de nombreux cas, comme par exemple [Her90, BD94, FD94, RL94], l'ordonnancement est équitable par principe. Il sélectionne régulièrement chaque processus, parfois même suivant une période fixe, et la question de l'utilité de l'équité ne se pose pas.

En revanche, il existe des situations dans lesquelles on ne veut ou ne peut pas supposer que l'ordonnancement est équitable. Dans ce cas, on a besoin d'algorithmes qui fonctionnent même dans l'hypothèse où l'on choisit (ou lorsqu'on est contraint) d'ignorer indéfiniment un même processus qui peut effectuer une action. Nous allons présenter dans ce chapitre quelques unes de ces situations où l'on n'a pas de système équitable. Une utilisation de l'algorithme du dîner des philosophes sans équité pour encoder le π-calcul synchrone dans le π-calcul probabiliste asynchrone est proposée à la section 6.1. Ensuite, à la section 6.2, nous décrivons deux autres situations dans lesquelles le problème de conserver ou non l'équité se pose : les algorithmes sur internet, et le cadre de la tolérance aux pannes.

6.1 Dîner des philosophes et π-calcul

Indépendamment de notre travail sur le dîner des philosophes probabilistes, O. Mihaela Herescu et Catuscia Palamidessi ont également travaillé sur ce même problème (voir [HP01, PH02]), dans un but *a priori* totalement différent : permettre l'encodage du π-calcul synchrone dans le π-calcul asynchrone probabiliste.

Le π-calcul ([MPW92]) est un langage de spécification qui a été développé dans le but de représenter et d'analyser des systèmes mobiles, c'est-à-dire des systèmes dont la topologie peut changer de manière dynamique. Il est basé sur l'algèbre de processus CCS, en lui ajoutant de la mobilité. Le problème lié au π-calcul (synchrone) réside dans la difficulté de son implantation. Certains mécanismes du π-calcul nécessitent la résolution d'un problème de consensus distribué, pour lequel on sait qu'il n'existe que des solutions probabilistes.

Il existe une autre version de ce calcul, le π-calcul asynchrone, qui a comme avantage d'être plus facilement implantable, mais dont l'inconvénient majeur est que son expressivité ne lui permet pas de modéliser une assez large gamme de systèmes distribués.

Dans le but d'augmenter l'expressivité du π-calcul asynchrone, Herescu et Palamidessi on introduit dans [HP00] une extension probabiliste, le π-calcul probabiliste asynchrone, basé sur les automates probabilistes étudiés par Segala et Lynch [SL95].
Un des intérêts du π-calcul asynchrone probabiliste est qu'il distingue les aspects probabiliste et non-déterministe. Les probabilités sont confinées dans les choix du processus, alors que tout le non-déterminisme réside dans les choix du démon extérieur. Ceci permet donc de représenter le cas où le démon se comporte comme un adversaire et essaie d'empêcher le processus d'atteindre ses objectifs.

Herescu et Palamidessi se sont donc intéressées à l'algorithme du dîner des philosophes probabilistes afin de réaliser l'encodage du π-calcul synchrone dans le π-calcul asynchrone probabiliste. Du fait que le π-calcul asynchrone n'impose pas d'hypothèse d'équité entre les différents processus, les auteurs se sont donc intéressés à une variante sans équité du dîner des philosophes, considérant ainsi dans un travail indépendant le même algorithme que celui étudié dans cette thèse.

Afin de comprendre un peu mieux en quoi l'algorithme du dîner des philosophes intervient dans cet encodage, il convient tout d'abord de présenter en quoi consistent les différentes formes de π-calcul.

6.1.1 Différents types de π-calcul

De manière informelle, le π-calcul décrit le comportement de processus, à l'aide de préfixes comme l'envoi ou la réception de message sur un canal, ou d'opérations comme la mise en parallèle, le choix entre plusieurs processus.

Plus précisément, considérons un ensemble dénombrable de noms de canaux $x, y, ...$ et un ensemble dénombrable de noms de processus $X, Y, ...$ L'ensemble des préfixes et l'ensemble des processus du π-calcul sont définis par la syntaxe suivante :

$$\begin{array}{rcll} \text{Préfixes} & \alpha & ::= & x(y) \mid \bar{x}y \mid \tau \\ \text{Processus} & P & ::= & \sum_i \alpha_i.P_i \mid \nu x P \mid P|P \mid X \mid rec_X P \end{array}$$

Dans cette syntaxe, les préfixes peuvent être interprétés de la manière suivante :
- $x(y)$ signifie que le processus reçoit le nom y sur le canal x,
- $\bar{x}y$ qu'il envoie le nom y sur le canal x,
- τ représente toute action silencieuse (pas de communication).

Le processus $\sum_i \alpha_i.P_i$ représente un choix : un seul des $\alpha_i.P_i$ va s'exécuter. On représente par 0 la somme vide (appelée *inaction*), et par $\alpha.P$ la somme à un élément.
Le processus $P_1|P_2$ consiste en la mise en parallèle des processus P_1 et P_2. Les composantes peuvent alors agir indépendamment ou se synchroniser (sur un envoi/réception de message).
Le symbole νx est l'opérateur de restriction. Il permet de définir un nouveau nom de canal, ici x, qui n'est pas connu des autres processus au moment de sa création. On remarque que, pour le processus $\nu x P$, si P est de la forme $P_1|P_2$, alors les deux processus en parallèle P_1 et P_2 peuvent communiquer au moyen du canal x.
Le processus $rec_X P$ représente un processus X défini de manière récursive : la définition de X peut contenir des occurrences de X.

Le but du travail d'Herescu et Palamidessi était d'encoder le π-calcul synchrone décrit ci-dessus dans une forme de π-calcul asynchrone[1]. La syntaxe de cette forme restreinte de

[1]En fait, le π-calcul asynchrone ne comporte même pas d'opérateur de choix (somme), mais Nestmann et Pierce [NP96] ont montré que le π-calcul décrit ici n'est pas plus expressif que le "véritable" π-calcul asynchrone.

π-calcul est la suivante :

$$\begin{array}{rrcl} \text{Préfixes sans envoi} & \alpha & ::= & x(y) \mid \tau \\ \text{Processus} & P & ::= & \bar{x}y \mid \sum_i \alpha_i.P_i \mid \nu x P \mid P|P \mid X \mid rex_X P \end{array}$$

La différence avec le π-calcul synchrone est qu'à présent l'opération de choix $\sum_i \alpha_i.P_i$ est restreinte aux préfixes sans envoi de message. Ainsi un processus qui envoie un message ne fait que ça, puis se termine. Comme on ne permet plus de termes $\bar{x}y.P$ dans une somme, le processus ne peut pas envoyer un message, attendre qu'un autre processus puisse se synchroniser en réception, puis continuer *via* P. C'est pourquoi on qualifie cette variante d'asynchrone.

Comme Palamidessi a montré dans [Pal97] qu'il était impossible "d'encoder le π-calcul synchrone dans le π-calcul asynchrone de manière uniforme et en préservant une sémantique raisonnable", Herescu et Palamidessi ont défini dans l'article [HP00] une nouvelle forme de π-calcul : le *π-calcul probabiliste asynchrone*, dans lequel elles ont montré (voir [PH02]) qu'il était possible d'encoder le π-calcul synchrone.

Le π-calcul probabiliste asynchrone est similaire au π-calcul asynchrone à une chose près : on modifie l'opérateur de choix (sans envoi de messages) $\sum_i \alpha_i.P_i$ en lui ajoutant des probabilités :

$$\sum_i p_i \alpha_i.P_i$$

Les p_i sont ici des probabilités dans l'intervalle $]0,1]$, de somme 1, et le choix entre les différents $\alpha_i.P_i$ se fait de manière probabiliste. Le processus va donc se continuer en P_i après avoir effectué la réception ou l'action silencieuse α_i, et ce avec probabilité p_i.

6.1.2 Encodage

Considérons à présent l'encodage. Comme les opérations de mise en parallèle, de restriction et de définition par récurrence existent dans le π-calcul probabiliste asynchrone, elles sont encodées de manière naturelle.

Le problème se pose pour l'opérateur de choix : lorsqu'on a, dans un choix en π-calcul synchrone, un terme du type $\bar{x}y.P_i$, comme ce genre de terme n'existe pas en π-calcul asynchrone, on va devoir le transformer en une composition parallèle d'un envoi de message d'une part, et d'un processus contenant P_i d'autre part, c'est-à-dire $\bar{x}\langle y, ..., \rangle \mid ...P_i$. Seulement, il ne faut pas que P_i s'exécute, dans les deux circonstances suivantes : quand le message n'a pas encore été envoyé, ou quand c'est un autre des $\alpha_j P_j$ qui a été choisi. Pour cela, on va faire en sorte que P_i ne s'exécute que lorsque l'envoi de message s'est synchronisé et qu'il est le seul processus à s'exécuter pour son opérateur de choix.

La notion de verrous

Le principe est le suivant : à chaque opérateur de choix est associé un *verrou* (*lock* en anglais) dit "principal" et, pour qu'un envoi/réception de message s'exécute, le processus correspondant à la réception (que l'on va appeler P_R) doit obtenir les deux verrous principaux, celui de son propre choix (appelé verrou *local*), et celui du choix de l'envoi correspondant (appelé verrou *distant*).

Comme il n'y a qu'un seul verrou principal par opérateur de choix, un processus qui obtient les deux verrous est sûr qu'il est le seul qui va pouvoir se synchroniser, et qu'aucun autre processus, ni dans le choix local ni dans le choix distant n'a été choisi avant lui ou son homologue P_E.

Lorsque P_R obtient ces deux verrous, il donne l'ordre à tout les autres processus $\alpha_j P_j$ de son choix local et du choix (distant) de P_E de s'arrêter, il poursuit son exécution et envoie au processus P_E un message lui signifiant que la synchronisation s'est effectuée et qu'il peut continuer.

Ainsi un envoi de message $\bar{x}y.P$ est, informellement changé en

$$\nu a(\bar{x}\langle a, ..., y\rangle \mid a(b).si\ b\ alors\ P\ sinon\ 0).$$

On crée donc un nouveau canal, a, dont le nom est envoyé (entre autres) sur le canal x en même temps que y. En parallèle un autre processus attend un message, b, sur le canal a, si ce message est $vrai$ alors P s'exécute, sinon il s'arrête. Comme a est un nouveau nom de canal, il n'est connu par aucun processus, à part P_R et celui qui va recevoir le message $\langle a, ..., y\rangle$, envoyé sur le canal x. Or P_R est le processus qui va essayer de se synchroniser avec P_E. Le processus P_R va donc essayer d'obtenir les deux verrous. S'il y parvient, il envoie le message $vrai$ à P_E qui va continuer son exécution par P. Sinon, il peut soit décider de tenter à nouveau d'obtenir les verrous, soit abandonner si un autre processus a été choisi à la place de P_R (et a réussi à obtenir les deux verrous pour se synchroniser).

Pour obtenir les verrous, on a besoin d'un algorithme symétrique, totalement distribué permettant à un processus d'obtenir deux verrous à la fois. Pour cela Herescu et Palamidessi ont choisi d'utiliser un algorithme similaire à celui du dîner des philosophes probabilistes. Ici les processus de réception de message jouent le rôle des philosophes, et les verrous le rôle des baguettes.
Dans le but d'avoir des algorithmes qui fonctionnent même en présence d'un ordonnancement adverse, les auteurs ont considéré la même variante que la nôtre de l'algorithme du dîner des philosophes probabilistes.

L'encodage d'un processus du type $x(y).P_i$ est assez complexe et correspond à la version en π-calcul de l'algorithme du dîner des philosophes probabilistes sans équité, avec deux principales particularités : premièrement lorsqu'un processus réussit à se synchroniser en envoi/réception, il envoie un message aux autres processus du choix local et du choix distant pour qu'ils s'arrêtent, et deuxièmement la première étape consiste à essayer d'obtenir un verrou auxiliaire h, décrit dans la section suivante.

Une description formelle et complète de l'encodage du π-calcul dans le π-calcul probabiliste asynchrone est donnée dans [PH02].

Problèmes de topologie

La différence majeure avec notre étude est que la topologie n'est ici pas un anneau, mais peut être plus compliquée. Une baguette (appelée dans ce cas verrou) n'est plus forcément partagée par deux processus, mais potentiellement par plus : tous les $\alpha_i.P_i$ du même opérateur de choix correspondant à des envois ou des réceptions partagent le même verrou. Ceci ne pose pas de problèmes, sauf quand le graphe contient deux cycles qui sont reliés entre eux, comme illustré dans la figure 6.1.

Pour résoudre ce problème, Herescu et Palamidessi ont introduit un deuxième type de verrou, les verrous auxiliaires. Il en existe un par opérateur de choix contenant des envois de messages et, avant d'essayer d'obtenir les deux verrous principaux (local et distant) en même temps, un processus correspondant à une réception doit obtenir le verrou auxiliaire du choix

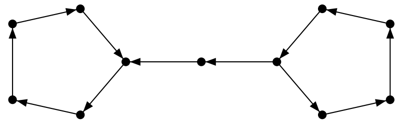

FIG. 6.1 – Un graphe contenant deux cycles reliés

de l'envoi avec lequel il essaie de se synchroniser. Lorsque des processus souhaitant réaliser une réception de message ont obtenu des verrous auxiliaires, on obtient un graphe où les sommets représentent les verrous principaux et où il y a un arc entre les verrous v et v' si v est le verrou local, v' le verrou distant correspondant, et le processus correspondant à cet arc a réussi à acquérir le verrou auxiliaire h de son partenaire.

Comme il y a au plus un verrou auxiliaire par opérateur de choix, le degré entrant des sommets du graphe ainsi construit (nombre d'arêtes arrivant dans un sommet) est au plus égal à un. Ceci entraîne que l'on ne peut avoir deux cycles reliés dans ce graphe, sinon, comme dans la figure 6.1, on aurait un sommet de degré entrant au moins égal à deux.

Dans [HP01], Herescu et Palamidessi ont montré que, lorsqu'on a une topologie sans cycles disjoints, on peut montrer la propriété de progrès en utilisant l'algorithme du dîner des philosophes probabilistes. On peut donc ici prouver la propriété de progrès qui assure qu'avec probabilité 1, un processus correspondant à la réception va obtenir les deux verrous et donc se synchroniser.

Une preuve du progrès de l'algorithme sans équité, d'un principe différent de celle présentée ici, est donnée dans la thèse de Herescu [Her02]. Elle se fonde également sur un motif particulier, appelé *paire de processus adjacents*, qui ressemble à notre notion de jeton. La preuve est basée sur plusieurs lemmes et montre, en plusieurs étapes, que l'on va atteindre un état où un des philosophes mange, et ce avec probabilité 1 et quel que soit l'ordonnancement.

6.2 Autres domaines concernés

6.2.1 Internet

Lorsque l'on s'intéresse à des algorithmes relatifs au réseau Internet, et plus particulièrement relatifs à l'acheminement de données, l'équité est une question importante. Un des points de vue est de chercher des algorithmes équitables d'allocation de bande passante, comme dans [BM01], afin d'obtenir une utilisation optimale des ressources du réseau.
Au contraire, d'autres travaux, comme par exemple [GRK99] ou [CO98] se penchent sur un principe d'utilisation différenciée des ressources. Selon le statut de l'utilisateur, il dispose de plus ou moins de ressources réseau pour faire circuler ses données. Ce principe permet alors de donner aux utilisateurs des services en fonction du prix qu'ils sont prêts à y mettre.

Ce dernier point de vue montre que, en ce qui concerne les algorithmes fonctionnant sur de gros réseaux, comme Internet, le besoin de l'équité n'est pas toujours évident. Certains travaux, comme ceux cités ici, abandonnent ou nuancent cette hypothèse d'équité en fonction des objectifs à atteindre.

6.2.2 Tolérance aux pannes

Dans le cadre des systèmes distribués, on peut faire différentes hypothèses, plus ou moins réalistes, sur le système selon ce que l'on cherche à prouver.

C'est le cas par exemple lorsqu'on considère des communications par échange de messages, et que l'on s'intéresse au niveau de fiabilité des canaux. Différentes hypothèses peuvent être faites, comme des canaux totalement fiables, dans lesquels tout message envoyé est reçu par le destinataire. On peut aussi supposer que les messages peuvent être perdus et ce de façon arbitraire. Enfin, certains travaux considèrent les canaux fiables avec une certaine probabilité. Les pertes de messages se font alors de manière probabiliste.

De la même manière, la plupart des algorithmes considèrent que les différents processus sont fiables, qu'ils vont tous réaliser leur tâche de manière satisfaisante. Si, par contre, on considère un modèle où ces processus peuvent tomber en panne ou décider de se bloquer, et ne plus répondre aux stimuli extérieurs, il devient déraisonnable de considérer que le démon est équitable à savoir que chaque processus qui est activable va effectuer infiniment souvent une action. À partir du moment où un processus se bloque, le démon ne pourra plus décider de le faire agir.

Si, lorsqu'un processus se bloque ou tombe en panne, il se met dans un état d'erreur, dans lequel aucune évolution n'est possible, alors il ne sera plus jamais activable et ne posera pas de problème pour l'équité. Si par contre le processus reste dans le même état tout en ne pouvant plus en changer, pour tout observateur extérieur ce processus ne sera pas différenciable d'un autre processus potentiellement activable. C'est pourquoi il faut envisager des ordonnancement qui peuvent ignorer indéfiniment un processus activable.

Pour certains problèmes, la propriété d'équité est cruciale, à savoir que, sans équité, on ne peut assurer par exemple la convergence. Ces systèmes ne pourront donc pas fonctionner si l'un des processus tombe définitivement en panne. Par contre, comme par exemple dans le cas du dîner des philosophes, il est possible, en modifiant quelque peu l'algorithme, de se ramener à une situation où l'équité n'est plus indispensable.

Ainsi, si avec notre variante on suppose qu'un ou plusieurs philosophes "s'endort" dans l'état H, simulant ainsi une panne, notre algorithme va continuer d'assurer la propriété de progrès. En fait la propriété de progrès sera conservée tant qu'au moins un philosophe reste "éveillé", et qu'un philosophe ne peut garder une baguette lorsqu'il s'endort.

De même, les algorithmes d'élection et de consensus prennent en compte la possibilité qu'un processus s'arrête, ou ne participe pas au vote (voir section 3.1.4).

L'absence d'équité permet donc de décrire des algorithmes tolérants aux pannes, dans le cas où l'on permet l'arrêt définitif d'un processus.

Troisième partie

Convergence de systèmes paramétrés déterministes

Chapitre 7

Preuve de convergence par surréductions

Dans le chapitre 3, nous avons déjà introduit les notions de règles et de système de réécriture dans le cas où la topologie du système distribué est un anneau. Cependant, ces notions ne nous servaient en fait qu'à représenter de manière simple et compacte les actions possibles pour les processus. Dans cette partie, nous allons vraiment nous intéresser à la réécriture comme outil de preuve de convergence de systèmes distribués.

Ce chapitre présente des travaux effectués par Joffroy Beauquier, Béatrice Bérard, Laurent Fribourg et Frédéric Magniette et publiés dans l'article [BBFM01], ainsi que l'outil qui a été réalisé en se basant sur ce travail. Certains points ont été modifiés pour mieux convenir au contexte qui va nous servir à présenter le chapitre suivant. Il était important de présenter ces résultats car, outre leur intérêt propre, ils ont également été le point de départ de notre travail sur l'automatisation de preuve de convergence.

Le plan du chapitre est le suivant. Dans un premier temps sont présentées à la section 7.1 les notions relatives aux systèmes de réécriture qui vont être utiles dans la suite de l'étude. La section 7.2 donne une définition de l'auto-stabilisation vers un ensemble, dans le cas précis des systèmes clos. La section 7.3 introduit la notion de dérivations du premier ordre, c'est-à-dire contenant des variables. Un théorème permettant de prouver l'auto-stabilisation en considérant des dérivations du premier ordre particulières est ensuite donné à la section 7.4. La section 7.5 revient sur la notion de φ-algorithmes présentée à la section 4.3, et explique comment elle permet dans ce contexte de simplifier les preuves. La notion de *schéma*, permettant de regrouper des configurations et d'essayer de se ramener à un système généralisé fini afin de prouver l'auto-stabilisation, est introduite à la section 7.6. Cette méthode de preuve a été implantée dans l'outil Poulet, qui est brièvement présenté à la section 7.7. Enfin la section 7.8 illustre la méthode en présentant la preuve d'auto-stabilisation de deux algorithmes tirés de la littérature.

7.1 Réécriture et réductions closes

Comme dans le chapitre sur les systèmes distribués, nous allons considérer un système, que nous appellerons $\mathcal{S}$. Ce système est composé d'une chaîne de processus, numérotés de 0 à $N-1$, dont les états possibles appartiennent à un alphabet Σ. Les transitions du système sont décrites au moyen de règles de réécriture. Dans les exemples que nous présenterons dans ce chapitre et le suivant, ces règles impliqueront deux ou trois lettres consécutives.

Comme annoncé à la section 3.4, nous allons considérer deux processus particuliers dans la chaîne : celui le plus à gauche appelé bottom, et celui le plus à droite appelé top. Pour effectuer cette distinction, il ne suffit plus de décrire les règles sous la forme $u \to v$, car on a à présent besoin de préciser le contexte, pour savoir où, dans le mot, a lieu la réécriture. C'est pourquoi, en plus de l'alphabet Σ, on va utiliser un ensemble de *variables du premier ordre* $\mathcal{V} = \{W, X, Y\}$ pour représenter de manière symbolique un facteur, c'est à dire un ensemble de lettres contiguës.

Pour une présentation ainsi qu'une étude des systèmes de réécriture, on peut se référer à [DJ90, BO93].

Définition 7.1. *On appelle* mot clos *("ground word" en anglais) tout élément de* Σ^**, c'est-à-dire tout mot sans variable. Par convention,* ε *désigne le mot vide.*
Un mot du premier ordre *("open word" en anglais) est un mot avec variable(s), à savoir un élément de* $(\Sigma \cup \mathcal{V})^*$*.*

Comme annoncé à la section 3.4, on va assimiler l'état local d'un processus à une lettre, et représenter les configurations du système par des mots. Une variable de $\mathcal{V}$ va servir à représenter un ensemble fini de processus consécutifs, dont on ne précise pas l'état. Un mot du premier ordre sert donc à décrire un ensemble de configurations pour lequel on ne connaît pas l'état de certains processus. Dorénavant, le terme de *mot* sera employé de manière indifférenciée pour désigner des mots clos et du premier ordre.

Notations : Dans la suite, les lettres t, u et v (avec ou sans indice) désigneront des mots, les lettres a, b, c et d (avec ou sans indice) désigneront des lettres de Σ, et les capitales W, X et Y désigneront des variables.

Une *substitution* est une fonction θ de $\mathcal{V}$ dans $(\Sigma \cup \mathcal{V})^*$, telle que $\theta(W) = W$ pour toutes les variables, excepté un ensemble fini noté $Dom(\theta)$. Elle peut donc se représenter par un ensemble fini de paires du type $\{W/\theta(W)\}_{W \in Dom(\theta)}$. Lorsque $Dom(\theta) = \emptyset$, la substitution ne modifie aucune variable et correspond donc à la fonction identité, notée *id*.
Cette fonction s'étend de façon naturelle à l'ensemble des mots par $\theta(\alpha_1 ... \alpha_n) = \theta(\alpha_1) ... \theta(\alpha_n)$ (pour des α_i appartenant à $\Sigma \cup \mathcal{V}$) avec, par convention, pour toute lettre $a \in \Sigma$, $\theta(a) = a$. L'image $\theta(t)$ (notée aussi $t\theta$) d'un mot t par l'application θ est alors appelée *instance* du mot t.
Une substitution est dite *close* si $\theta(W)$ est dans Σ^* pour tout $W \in Dom(\theta)$, et on appelle *instance close* d'un mot t du premier ordre toute image de t par une substitution close.
En se basant sur la notion d'instance, on peut considérer un mot du premier ordre $abWcb$ comme la représentation symbolique de toutes ses instances closes possibles $ab\Sigma^*cb$.

Remarque : Dans la suite de cette partie, nous allons représenter les (ensemble de) configurations par des mots à une seule variable W. Ainsi, lorsqu'on va appliquer une substitution close θ avec $W \in Dom(\theta)$, on va obtenir un mot clos.

Dans la section 3.4, nous avons décrit les transitions possibles du système à l'aide de règles de réécriture, mais en se concentrant sur le cas où le système est symétrique, c'est-à-dire où chaque processus est semblable aux autres. Ici, pour présenter dans un premier temps le travail de Beauquier *et al.* et ensuite notre travail pour automatiser cette vérification, nous avons besoin de considérer le cas où certaines machines sont différentes et nécessitent des règles particulières.

Un système de réécriture $\mathcal{S}$ sera décrit par un ensemble de règles préservant la longueur, réparties en trois sous-ensembles[1] : $Top_{\mathcal{S}}$, $Bottom_{\mathcal{S}}$ et $Middle_{\mathcal{S}}$. Plus précisément, soient l, r (resp. l_i, r_i avec $i \in \{1,2\}$) des mots clos non vides de même longueur, on a :

- $Middle_{\mathcal{S}}$ se compose des règles de la forme $XlY \to XrY$;
- $Top_{\mathcal{S}}$ se compose des règles de la forme $Xl \to Xr$ ou $l_1Xl_2 \to r_1Xr_2$;
- $Bottom_{\mathcal{S}}$ se compose des règles de la forme $lX \to rX$.

Les deux cas possibles pour les règles de $Top_{\mathcal{S}}$ représentent respectivement le cas où les deux extrémités du réseau sont isolées (topologie linéaire), et le cas où elles communiquent (topologie en anneau).

Pour ne pas confondre les règles de $Bottom_{\mathcal{S}}$ et les règles de $Middle_{\mathcal{S}}$, il ne faut pas que les règles de $Middle_{\mathcal{S}}$ modifient la lettre la plus à gauche. Il faut donc parfois préciser que, dans les règles de $Middle_{\mathcal{S}}$, la variable X ne peut pas représenter le mot vide. De même, dans le cas de la topologie linéaire, pour ne pas confondre les règles de $Middle_{\mathcal{S}}$ et $Top_{\mathcal{S}}$, on doit supposer que la variable Y ne représente pas le mot vide.

Exemple 7.2. Considérons l'algorithme suivant, qui est une version simplifiée de l'algorithme de Beauquier-Debas [BD95] qui assure l'exclusion mutuelle auto-stabilisante sur un anneau. Cette version est obtenue en utilisant la simplification qui sera décrite à la section 7.5. L'alphabet est $\Sigma = \{0,1,2\}$ et l'ensemble des règles est $\mathcal{BD} = \{B_1, M_1, M_4, T_4\}$ (voir section 7.8.1 pour le système complet) :

$$\begin{array}{ll} B_1: & 12X \to 21X \\ M_1: & X10Y \to X01Y \text{ (avec } X \neq \varepsilon) \\ M_4: & X02Y \to X20Y \text{ (avec } X \neq \varepsilon) \\ T_4: & 2X1 \to 1X2 \end{array}$$

Ici T_4 est une règle de $Top_{\mathcal{BD}}$ qui permet aux deux extrémités de communiquer. On est donc dans le cas de la topologie en anneau. B_1 est une règle de $Bottom_{\mathcal{BD}}$ et M_1 et M_4 sont des règles de $Middle_{\mathcal{BD}}$. On ajoute la condition $X \neq \varepsilon$ pour les deux règles de $Middle_{\mathcal{BD}}$ pour ne pas qu'elles modifient la lettre la plus à gauche. ♣

Une *réduction close* sur un mot clos consiste à remplacer une instance du membre gauche de la règle par l'instance correspondante du membre droit de la règle. Formellement, étant donnés deux mots clos t et t', on dit qu'on peut réduire t en t' au moyen de la règle $M : XabY \to Xa'b'Y$ appartenant à $Middle_{\mathcal{S}}$, et on note $t \underset{M}{\longrightarrow} t'$ s'il existe $u \in \Sigma^+$ et $v \in \Sigma^*$ tels que $t = uabv$ et $t' = ua'b'v$. Si la topologie est linéaire, il faut de plus supposer que $v \in \Sigma^+$. Le mot clos t est bien une instance du membre gauche de la règle M pour la substitution $\{X/u, Y/v\}$.

La notion de réduction close s'applique également lorsque $t = t_1Wt_2$ est un mot à une variable. La variable W est alors considérée comme une constante et ajoutée à Σ, et la réduction *via* M consiste à remplacer un motif ab en $a'b'$ dans une des parties closes t_1 ou t_2 de t.

La notion de réduction close d'un mot t (clos ou non) se définit de manière similaire pour une règle de $Top_{\mathcal{S}}$ (resp. $Bottom_{\mathcal{S}}$).

[1]Les trois types de règles considérées ici ne correspondent pas exactement au contexte de l'article [BBFM01], où ce sont les règles de $Bottom_{\mathcal{S}}$ qui assurent le cas échéant la communication entre les deux extrémités du réseau. Ce choix nous sera utile dans le chapitre suivant, et est déjà présenté ici par souci de cohérence.

Exemple 7.3. Reprenons le système $\mathcal{BD}$ de l'exemple 7.2, et considérons le mot clos $t = 1201$. Le mot t est bien une instance du membre gauche de la règle B_1 : $12X \rightarrow 21X$ (pour la substitution $\{X/01\}$), et on a donc $t \underset{B_1}{\longrightarrow} t' = 2101$.
Le mot du premier ordre $u = 12W2$ est également une instance du membre gauche de B_1 pour la substitution $\{X/W2\}$. On peut donc lui appliquer une réduction close *via* B_1, ce qui donne $u \underset{B_1}{\longrightarrow} u' = 21W2$. ♣

Lorsque l'on ne veut pas préciser le nom de la règle qui a été appliquée, étant donnés un système de réécriture $\mathcal{S}$ et deux mots t et t', on dit qu'une réduction close *via* $\mathcal{S}$ s'applique à t et donne t', et on note $t \underset{\mathcal{S}}{\rightarrow} t'$ si $t \underset{R}{\rightarrow} t'$ pour une certaine règle $R \in \mathcal{S}$.

On reprend la notion d'ensemble *fermé* pour un système de réécriture, décrite à la section 3.6.4.

Exemple 7.4. L'ensemble $\mathcal{L}_{\mathcal{BD}} = 20^*10^* \cup 10^*20^*$ est fermé pour le système $\mathcal{BD}$ présenté à l'exemple 7.2. ♣

Définition 7.5. *Une* dérivation close *est une suite (éventuellement infinie) de réductions consécutives sur des mots clos.*

Cette définition correspond à celle d'exécution définie à la section 3.5.1. La seule différence est qu'ici une dérivation n'est pas *a priori* supposée infinie.

Définition 7.6. *On dit qu'un système de réécriture $\mathcal{S}$ est* nœtherien*, ou qu'il* termine *si toute dérivation close utilisant des règles de $\mathcal{S}$ est finie.*
Un système $\mathcal{S}$ est dit équitable *si toute dérivation close infinie suivant $\mathcal{S}$ réécrit infiniment souvent chaque position.*

La définition de système équitable reprend l'idée annoncée à la section 3.5.3 selon laquelle ce n'est plus l'ordonnancement ou le démon qui est équitable, mais le système lui-même, qui ne permet l'existence que d'ordonnancements équitables.
Cette définition a un sens du fait que toutes les règles préservent la longueur. Si les règles changeaient la longueur des mots, la position d'un même processus pourrait changer au cours d'une dérivation et le fait de réécrire infiniment souvent chaque position ne signifierait plus rien pour les processus eux-mêmes.

Enfin, nous pouvons définir la notion de cycle, qui va être utile pour donner dans ce contexte une caractérisation de l'auto-stabilisation.

Définition 7.7. *Une dérivation close est dite* cyclique *si elle est de la forme $t_1 \underset{\mathcal{S}}{\rightarrow} t_2 \underset{\mathcal{S}}{\rightarrow} \ldots \underset{\mathcal{S}}{\rightarrow} t_n$ avec $t_n = t_1$.*

Les notions de dérivation cyclique et de système nœtherien ne sont pas indépendantes, comme on peut le voir dans le lemme suivant.

Lemme 7.8. *Soit $\mathcal{S}$ un système de réécriture composé de règles préservant la longueur. Le système $\mathcal{S}$ est nœtherien si et seulement si il n'admet pas de dérivation close cyclique.*

Démonstration. Tout d'abord, s'il existe un cycle pour le système $\mathcal{S}$, alors en l'itérant on obtient une dérivation infinie et le système n'est pas nœtherien.
Réciproquement, supposons que le système ne soit pas nœtherien. Par définition, il existe une

dérivation infinie $t_1 \to ... \to t_n ...$ pour $\mathcal{S}$. Comme l'alphabet Σ est fini et comme les règles de $\mathcal{S}$ préservent la longueur, le long d'une dérivation close il y a au un nombre fini (au plus Σ^N où N est la longueur du mot t_1) de mots différents. Il existe donc deux indices $i < j$ tels que $t_i = t_j$, et par conséquent une dérivation cyclique $t_i \to ... \to t_j$ pour $\mathcal{S}$. □

Pour illustrer ces définitions, reprenons l'exemple 7.2, page 121.

Exemple 7.9. Considérons le système $\mathcal{BD}$ où, pour augmenter la lisibilité, pour chaque réduction les lettres qui vont être changées sont écrites en gras. On a :

$$t_1 = 2\mathbf{10}0 \underset{M_1}{\longrightarrow} 20\mathbf{10} \underset{M_1}{\longrightarrow} \mathbf{2}00\mathbf{1} \underset{T_4}{\longrightarrow} 10\mathbf{02} \underset{M_4}{\longrightarrow} 1\mathbf{02}0 \underset{M_4}{\longrightarrow} \mathbf{12}00 \underset{B_1}{\longrightarrow} 2100 = t_1.$$

Il existe donc une dérivation infinie et cyclique qui consiste à itérer le cycle ci-dessus. On en déduit que le système $\mathcal{BD}$ n'est pas nœtherien.
Par contre, le système $\mathcal{BD} - T_4$ est nœtherien. Considérons la fonction ψ qui associe à tout mot clos $t = a_0 \cdots a_{N-1}$ la valeur $\psi(t) = \sum_{i \in S_1}(N - i) + \sum_{j \in S_2} j$ où S_1 (resp. S_2) désigne l'ensemble des positions i (resp. j) telles que $a_i = 1$ (resp. $a_j = 2$). Il est facile de voir que l'application de la règle M_1 (ou M_4) fait décroître ψ de 1, et que B_1 fait décroître ψ de 2. Comme la fonction $\psi(t)$ ne peut prendre que des valeurs positives et que toutes les règles de $\mathcal{BD} - T_4$ la font décroître, on ne peut pas appliquer infiniment des règles de $\mathcal{BD} - T_4$ qui est donc nœtherien. ♣

7.2 Auto-stabilisation vers un ensemble clos

À présent que les notions de base sont introduites, on peut donner une définition d'auto-stabilisation restreinte dans ce contexte de systèmes de réécriture. Comme annoncé à la section 3.6.4, le type d'auto-stabilisation qui va nous intéresser est assez restreint. Nous allons l'appeler auto-stabilisation vers un ensemble.

Définition 7.10. *Un système $\mathcal{S}$ est dit* auto-stabilisant vers un ensemble *de configurations $\mathcal{L}$ si les trois conditions suivantes sont vérifiées :*

1. *Chaque mot clos n'appartenant pas à $\mathcal{L}$ est réductible ;*
2. *L'ensemble $\mathcal{L}$ est fermé pour le système $\mathcal{S}$;*
3. *Il n'y a pas de dérivation cyclique $t_1 \underset{\mathcal{S}}{\to} ... \underset{\mathcal{S}}{\to} t_n = t_1$ avec $t_1 \notin \mathcal{L}$.*

Les deux spécificités de cette définition sont d'une part que la spécification est particulière (rester dans l'ensemble $\mathcal{L}$) et d'autre part que, comme annoncé dans le titre de cette section, l'ensemble $\mathcal{L}$ est clos.

Dans tous les chapitres à venir, même quand ce ne sera pas précisé, nous allons utiliser uniquement cette définition d'auto-stabilisation.

7.3 Dérivations du premier ordre

Afin de prouver l'auto-stabilisation, on a besoin de calculer un ensemble vers lequel le système converge, c'est-à-dire par lequel toute dérivation va passer. Seulement, si on doit vraiment considérer toutes les dérivations, il est difficile de dégager un ensemble intéressant vers lequel toute dérivation converge. C'est pourquoi l'article [BBFM01] s'intéresse, grâce aux variables et aux mots du premier ordre, à restreindre les ensembles de dérivations à considérer, tout en en conservant suffisamment pour assurer la convergence.

7.3.1 Unification

Dans cette partie, pour étudier ces généralisations de dérivations, il n'est utile de considérer que des mots à une seule variable, c'est-à-dire de la forme uWv avec $W \in \mathcal{V}$ et $u, v \in \Sigma^*$.

Dans la section 7.1, nous avons vu le concept de réductions closes, dans le cas des mots à une variable. Au cours de ces réductions, la variable n'est pas changée. On peut donc considérer que cette réduction a été effectuée en utilisant la substitution $\theta = id$.

Cependant, on peut également s'intéresser à des substitutions $\theta \neq id$ qui rendent un mot réductible. On va d'abord appliquer une (ou plusieurs) substitution(s) à la règle d'une part et au mot à une variable d'autre part afin d'obtenir, en appliquant ces substitutions à l'un et à l'autre, deux instances identiques.

Définition 7.11. *Soient t et u deux mots. On appelle* unificateur *de t et u toute substitution μ telle que $\mu(t) = \mu(u)$.*

Dans le cas général, le problème de trouver des unificateurs pour un couple donné de mots du premier ordre est très complexe. Dans notre cas, comme on considère que les règles et configurations n'ont pas de variables en commun, et que chaque variable apparaît au plus une fois dans un mot, le problème est beaucoup plus simple.

En fait, pour ce qui va nous intéresser par la suite, on ne va considérer que des unificateurs minimaux, et qui ne créent pas de membre gauche de règle exclusivement à partir d'une variable. Ainsi, pour la règle $XabY \to Xa'b'Y$, on ne va pas considérer de substitution du genre $\{W/W'ab\}$ car ici les deux lettres du membre gauche de la règle ont été créées lors de la substitution.

Exemple 7.12. Si l'on considère la règle $M_1 : X10Y \to X01Y$ de l'exemple 7.2, page 121, le mot $t = 1W0$ peut être unifié au membre gauche $X10Y$ par, entre autres, les unificateurs suivants :

μ_1 : $\{W/W'1, X/1W', Y/\varepsilon\}$

μ_2 : $\{W/\varepsilon, X/\varepsilon, Y/\varepsilon, \}$

μ_3 : $\{W/W_1 10 W_2, X/1W_1, Y/W_2 0\}$

Le dernier de ces unificateurs, μ_3, ne va pas être considéré car il correspond à une unification ayant lieu à l'intérieur d'une variable et donc les deux lettres du membre gauche sont créées lors de la substitution. Dans ce cas précis, on ne va pas non plus considérer l'unificateur μ_2, du fait que la règle M_1 est du type *Middle* et que de ce fait la variable X ne peut pas être changée en ε (voir le système $\mathcal{BD}$ présenté à l'exemple 7.2, page 121).

On peut ensuite, après unification, appliquer la réduction suivante, correspondant à μ_1 : $1W'10 \to 1W'01$. ♣

Soit $R : XlY \to XrY$ une règle de *Middle*, avec $l = a_1...a_n$ où les a_i sont des lettres de Σ. Les substitutions $\theta \neq id$ utilisées pour faire des unificateurs minimaux du membre gauche de R et d'un mot sont d'une des trois formes suivantes :

- les substitutions à droite sont de la forme $\alpha_i : \{W/W'a_1...a_i\}$ pour $1 \leq i \leq n-1$;
- les substitutions à gauche sont de la forme $\beta_i : \{W/a_{i+1}...a_nW'\}$ pour $1 \leq i \leq n-1$;

– les substitutions closes sont de la forme $\gamma_{ij} : \{W/a_{i+1}...a_j\}$ pour $1 \leq i \leq j \leq n-1$, avec par convention $\gamma_{ij} : \{W/\varepsilon\}$ lorsque $i = j$.

On note D_R cet ensemble de *substitutions minimales* associées à R.
On peut définir un ensemble de substitutions similaires pour les règles de *Top* ou *Bottom*.

7.3.2 Surréduction

Nous allons à présent, toujours en suivant le formalisme de [BBFM01], décrire l'opération de *surréduction*, appelée *narrowing* en anglais (voir [DJ90]), dans ce cadre précis où l'on ne considère que des unificateurs minimaux et pas de substitution au sein d'une variable.

Une *surréduction* consiste en une étape d'unification minimale pour une certaine règle R avec la restriction décrite ci-dessus, suivie d'une étape de réduction close *via* cette même règle R. Elle est associée à la notion de "minimal reduction" utilisée dans [BBFM01] dont voici la définition.

Définition 7.13. *On dit que le mot du premier ordre t est* réductible de façon minimale *en u* via *la règle $R : \lambda \to \rho$ en utilisant la substitution $\theta \in D_R \cup id$, et on note $t\theta \underset{R}{\longrightarrow} u$ lorsque :*
- *soit $\theta = id$, t est une instance de λ et u est l'instance correspondante de ρ ;*
- *soit $R : XlY \to XrY$ est une règle de Middle, avec $l = a_1...a_n$, et $t = t_1Wt_2$ et alors on a les possibilités suivantes*
  - *le facteur t_2 commence par $a_{i+1}...a_n$ (avec $1 \leq i \leq n-1$), $\theta = \alpha_i$ et u est obtenu en remplaçant dans t le facteur $Wa_{i+1}...a_n$ par $W'r$;*
  - *le facteur t_1 termine par $a_1...a_i$ (avec $1 \leq i \leq n-1$), $\theta = \beta_i$ et u est obtenu en remplaçant dans t le facteur $a_1...a_iW$ par rW' ;*
  - *le facteur t_2 commence par $a_{j+1}...a_n$ et le facteur t_1 termine par $a_1...a_i$ (avec $1 \leq i \leq j \leq n-1$), $\theta = \gamma_{ij}$ et u est obtenu en remplaçant dans t le facteur $a_1...a_iWa_{j+1}...a_n$ par r ;*
- *soit R est une règle de Bottom ou Top, auquel cas on définit la réduction minimale de la même manière que pour les règles de Middle.*

Dans ce cas, on pourra dire de manière équivalente que u est obtenu à partir de t par une étape de *surréduction*, en utilisant la règle R et pour une substitution minimale θ, noté $t \rightsquigarrow_R u$ (en général la substitution θ est omise dans la notation).

Remarque : Pour pouvoir faire la différence entre les réductions closes et surréductions utilisant des substitutions non triviales, nous allons réserver dans la suite le terme de surréduction au cas où l'on utilise une substitution $\theta \neq id$.

Exemple 7.14. Si on considère la règle $M_1 : X10Y \to X01Y$ du système de réécriture $\mathcal{BD}$, et le mot du premier ordre, $t = 21W10$, la transformation $21W10 \rightsquigarrow_{M_1} 201W'10$ sera appelée surréduction (associée à la substitution $\{W/0W'\}$, mais la transformation de t en $21W01$ (qui a proprement parler est une surréduction associée à la substitution $\theta = id$) sera uniquement appelée réduction et notée $21W10 \underset{M_1}{\longrightarrow} 21W01$. ♣

La notation $t \rightsquigarrow_{\mathcal{S}} u$ signifie que $t \rightsquigarrow_R u$ pour une certaine règle R de $\mathcal{S}$. De même, on emploie la notation classique $\rightsquigarrow_{\mathcal{S}}^*$ pour désigner la fermeture symétrique et transitive de $\rightsquigarrow_{\mathcal{S}}$. Dans la suite, lorsque l'on voudra différencier les surréductions à droite (correspondant aux substitutions à droite), à gauche (correspondant aux substitutions à gauche pour les règles de

Middle), “bottom” (correspondant aux substitutions à gauche pour les règles de *Bottom*) ou closes (correspondant aux substitutions closes), on pourra les noter respectivement $\leadsto^{droite}$, $\leadsto^{gauche}$, $\leadsto^{bot}$ et $\leadsto^{clos}$.

Exemple 7.15. Considérons la règle M_4 : $X02Y \to X20Y$ et le mot $t = W200$. On peut appliquer une seule surréduction à t en utilisant M_4 : elle utilise la substitution $\{W/W'0\}$ et se note $t \leadsto_R^{droite} t' = W'2000$. ♣

7.3.3 Chaînes de réduction minimales issues de $Top_{\mathcal{S}}$

Comme annoncé précédemment, nous n'allons pas nous intéresser à toutes les dérivations possibles dans $\mathcal{S}$, mais à une classe particulière, celle des *chaînes top*.

Définition 7.16. *Les* chaînes de réduction minimales issues de $Top_{\mathcal{S}}$*, également appelées* chaînes top*, sont définies inductivement de la manière suivante :*
- *toute règle de $Top_{\mathcal{S}}$ est une chaîne top ;*
- *si C : $t_0 \to t_1 \to \ldots \to t_n$ est une chaîne top et si $t_n \leadsto u$* via *une règle R et une substitution minimale θ, alors $t_0\theta \to \ldots \to t_n\theta \underset{R}{\to} u$ est une chaîne top.*

Exemple 7.17. Considérons la règle T_4 du système $\mathcal{BD}$ présenté à l'exemple 7.2, page 121. Cette règle constitue une chaîne top $2W1 \underset{T_4}{\longrightarrow} 1W2$. On peut alors prolonger cette chaîne en effectuant des réductions minimales successives *via* la règle B_1, puis M_1, et obtenir la chaîne $220W''1 \underset{T_4}{\longrightarrow} 120W''2 \underset{B_1}{\longrightarrow} 210W''2 \underset{M_1}{\longrightarrow} 201W''2$. Les substitutions correspondantes sont ici successivement $\theta_1 = id$, $\theta_2 = \{W/2W'\}$ et $\theta_3 = \{W'/0W''\}$. Si on reprend la notation donnée dans la définition, cette même chaîne peut s'écrire :

$$2W1\,\theta_1\theta_2\theta_3 \underset{T_4}{\longrightarrow} 1W2\,\theta_2\theta_3 \underset{B_1}{\longrightarrow} 21W'2\,\theta_3 \underset{M_1}{\longrightarrow} 201W''2.$$

♣

Définition 7.18. *Une chaîne top (resp. une dérivation close) $t_0 \to \ldots \to t_n$ est dite* quasi-cyclique *s'il existe un indice $i < n$ tel que $t_i = t_n$ et, pour tout couple (j,k) d'indices distincts, strictement inférieurs à n, $t_j \neq t_k$.*

7.4 Auto-stabilisation au premier ordre

Grâce à cette caractérisation de dérivations particulières et la notion de chaînes quasi-cycliques, on peut donner une nouvelle caractérisation de l'auto-stabilisation (cf. théorème 5 de [BBFM01]).

Théorème 7.19. *Soit $\mathcal{S} = Middle_{\mathcal{S}} \cup Top_{\mathcal{S}} \cup Bottom_{\mathcal{S}}$ un système de réécriture, et $\mathcal{L}$ un ensemble de configurations. Supposons que*

1. *tout mot clos est réductible pour $\mathcal{S}$,*
2. *$\mathcal{L}$ est fermé pour $\mathcal{S}$, et*
3. *$\mathcal{S} - Top_{\mathcal{S}}$ est nœtherien,*

alors le système $\mathcal{S}$ est auto-stabilisant vers $\mathcal{L}$ si et seulement si il n'y a pas, dans $\mathcal{S}$, de chaîne top quasi-cyclique $t_0 \to \ldots \to t_n$ telle qu'il existe une instance u_n de t_n qui ne soit pas dans $\mathcal{L}$.

Autrement dit, pour un système auto-stabilisant, s'il existe une chaîne top quasi-cyclique $t_0 \to \ldots \to t_n$, alors toute instance close de t_n est dans $\mathcal{L}$.

La preuve de ce théorème, basée sur un résultat dû à Dershowitz, se trouve dans l'article [BBFM01]. Nous allons ici en donner les grandes lignes.

Tout d'abord, comme $\mathcal{S} - Top_{\mathcal{S}}$ est nœtherien, toute dérivation close infinie contient au moins une application d'une règle de $Top_{\mathcal{S}}$. Après avoir appliqué une telle règle, mettons $aXb \to a'Xb'$, on obtient un mot clos de la forme $a'ub'$, avec $u \in \Sigma^*$. Ainsi, l'ensemble des mots dérivés des mots du type $a'ub'$ est inévitable. Cependant, parmi ces dérivations, on va encore se restreindre, en utilisant un résultat dû à Dershowitz (voir [Der81]).

Dans un mot clos, on va distinguer une partie *active ! partie* et une partie *inactive*, qui vont évoluer le long d'une dérivation. Initialement, dans notre cas, toutes les lettres sont inactives. Après l'application de la première règle de $Top_{\mathcal{S}}$, seules les lettres ayant été modifiées par l'application de la règle de $Top_{\mathcal{S}}$ seront actives. Au cours de la dérivation, toute lettre qui à un moment devient active le reste. Pour une réduction donnée, si elle n'affecte que des lettres de la partie inactive, alors les parties actives et inactives ne changent pas. Si par contre on applique une règle qui réécrit une ou plusieurs lettres de la partie active, alors toutes les positions concernées par cette application de règle deviennent actives.

Exemple 7.20. Dans cet exemple, pour différencier les parties active et inactive, les lettres de la partie active seront écrites en caractères gras.
Considérons les règles du système $\mathcal{BD}$ de l'exemple 7.2, page 121. Comme elles modifient toujours deux lettres à la fois, on va adopter la convention suivante : lorsqu'on effectue une réduction qui modifie les lettres aux positions i et $i+1$, on va dire qu'on applique la règle à la position i.
Partant d'une instance close $t = \mathbf{1}0210\mathbf{2}$ du membre droit de la règle T_4, seules les lettres top et bottom sont actives.
Si l'on applique ensuite la règle $M_1 : X10Y \to X01Y$ à la position 3, comme les lettres affectées sont toutes les deux dans la partie inactive, la partie active ne s'étend pas. On obtient la réduction $t \to \mathbf{1}0201\mathbf{2}$. Si par contre on applique à t la règle $M_4 : X02Y \to X20Y$ à la position 4, une des deux lettres affectées est active, et l'autre inactive. La partie active va donc s'étendre et on obtient $t \to \mathbf{1}021\mathbf{20}$. ♣

L'intérêt de distinguer ces deux parties est double. D'une part, pour toute dérivation active $w_1 \to w_2 \to \ldots \to w_n$, c'est-à-dire ne comprenant que des réductions qui réécrivent (au moins) la partie active du mot, il existe une chaîne top $t_1 \to t_2 \to \ldots \to t_n$ et une substitution θ telle que $t_1\theta = w_1$, $t_2\theta = w_2$, ... $t_n\theta = w_n$. Autrement dit, toute dérivation active est une instance (close) d'une chaîne top. D'autre part, si on a une dérivation composée de réductions actives et inactives entremêlées, alors il va être possible, grâce à un lemme dit de “semi-commutation” de Dershowitz ([Der81]), de permuter ces réductions pour faire “remonter” les réductions inactives en début de dérivation, tout en conservant la même configuration de départ et la même configuration d'arrivée.

En utilisant le fait que le système $\mathcal{S} - Top_{\mathcal{S}}$ est nœtherien, Beauquier *et al.* montrent, en utilisant le lemme de semi-commutation, que toute dérivation close infinie ne contient qu'un nombre fini de réductions inactives. Ainsi, après ces commutations, on obtient une dérivation dont la première partie (finie) est inactive, et le reste est actif.

En mettant ensemble ces différents arguments, les auteurs prouvent que l'existence de cycles dans des dérivations closes est équivalente à l'existence de cycles dans les chaînes top, et qu'ainsi, en considérant uniquement ces chaînes top et en détectant les cycles éventuels hors de $\mathcal{L}$, on va pouvoir décider si le système est auto-stabilisant.

7.5 Retour aux φ-algorithmes

Comme nous l'avons fait dans la section 4.3 du chapitre sur les systèmes distribués probabilistes, Beauquier *et al.* ont considéré, lorsque c'est possible, une fonction φ sur les configurations, à valeurs dans un ensemble bien ordonné, et telle que φ ne croît jamais lors d'une réduction.

Supposons que la fonction φ est telle que l'on peut définir une version de $\mathcal{S}$, notée $\mathcal{S}'$, qui préserve φ, c'est-à-dire telle que :

$$t \underset{\mathcal{S}'}{\longrightarrow} u \iff (t \underset{\mathcal{S}}{\rightarrow} u \wedge \varphi(t) = \varphi(u)).$$

Alors il va être suffisant de considérer le système $\mathcal{S}'$ au lieu de $\mathcal{S}$. Autrement dit, le théorème 7.19 devient :

Théorème 7.21. *Soit $\mathcal{S} = Middle_{\mathcal{S}} \cup Top_{\mathcal{S}} \cup Bottom_{\mathcal{S}}$ un système de réécriture, et $\mathcal{L}$ un ensemble de configurations. Soit φ une fonction non croissante le long des dérivations et $\mathcal{S}' = Middle_{\mathcal{S}'} \cup Top_{\mathcal{S}'} \cup Bottom_{\mathcal{S}'}$ une version de $\mathcal{S}$ qui préserve φ. Supposons que*

1. *Tout mot clos est réductible pour $\mathcal{S}$;*
2. *L'ensemble $\mathcal{L}$ est fermé pour $\mathcal{S}$; et*
3. *Le système $\mathcal{S}' - Top_{\mathcal{S}'}$ est nœtherien ;*

alors le système $\mathcal{S}$ est auto-stabilisant vers $\mathcal{L}$ si et seulement si il n'y a pas, dans $\mathcal{S}'$, de chaîne top quasi-cyclique $t_0 \rightarrow ... \rightarrow t_n$ telle qu'il existe une instance u_n de t_n qui ne soit pas dans $\mathcal{L}$.

Cette simplification est possible car, comme les règles préservent la longueur, il existe un nombre fini de configurations t distinctes le long d'une dérivation (au plus $|\Sigma|^N$), d'où un nombre fini d'images $\varphi(t)$ distinctes. Il n'y a donc pas de suite infinie strictement décroissante pour φ, et toute dérivation infinie va se composer d'une première partie, finie, décroissante (pas forcément strictement), et d'une deuxième partie, infinie cette fois, mais constante pour φ . Cette deuxième partie va donc n'utiliser que des règles de $\mathcal{S}'$. Ainsi s'il y a un cycle, il ne va utiliser que des règles de $\mathcal{S}'$, et il suffit de détecter les exécutions quasi-cycliques pour $\mathcal{S}'$.

Exemple 7.22. Le système $\mathcal{BD}$, présenté à l'exemple 7.2, page 121 est une version du système de Beauquier-Debas [BD95], qui sera présenté intégralement à la section 7.8. Le système $\mathcal{BD}$ est constitué des règles de l'algorithme de Beauquier-Debas qui préservent une certaine fonction $\varphi_{\mathcal{BD}}$, qui compte le nombre de processus dans un état différent de 0.

De même, à la section 7.8, sera présenté un algorithme auto-stabilisant, variante de l'algorithme de Dijkstra à quatre états, créé par Ghosh ([Gho93]), dont la version originale est à deux lettres, mais donc la version conservant une certaine fonction $\varphi_{\mathcal{G}}$ comporte des règles à trois lettres. ♣

7.6 Schémas

Le fait d'introduire une fonction φ et de l'utiliser pour ne considérer que des chaînes top préservant φ permet de limiter grandement le nombre de dérivations à considérer. Cependant, même avec cette restriction, le nombre de chaînes à engendrer pour détecter les éventuels cycles ainsi que la longueur de ces chaînes sont en général infinis. C'est pourquoi Beauquier *et al.* introduisent une généralisation de la notion de chaîne, qui porte non plus sur des mots, mais sur des ensembles réguliers de mots.

Définition 7.23. *Un* schéma du premier ordre *S est un langage de la forme LWM où L et M sont des langages réguliers inclus dans Σ^*, et W est une variable ($\in \mathcal{V}$). Un* schéma clos *est un langage régulier $L \in \Sigma^*$. Le terme de* schéma *englobe les deux types de schémas décrits ci-dessus, clos et du premier ordre.*

Ce genre de généralisation est utilisé car il permet de représenter en une seule fois un ensemble de configurations, engendrées par exemple en appliquant un certain nombre de fois une même règle.

Exemple 7.24. Considérons à nouveau le système $\mathcal{BD}$, et plus spécialement les règles T_4 : $2X1 \to 1X2$ et $M_4 : X02Y \to X20Y$. Partant de T_4, si on applique la règle M_4 de manière répétée, on obtient des chaînes de la forme :
$2W1 \to 1W2$
$2W01 \to 1W02 \to 1W20$
$2W001 \to 1W002 \to 1W020 \to 1W200$
etc.
On peut remarquer que l'on a en bout de chaîne soit le mot $1W2$ lorsqu'on n'a pas appliqué M_4, soit un mot du type $1W20^j$ avec $j > 0$. On va alors rassembler toutes les configurations du type $1W20^j$ avec $j > 0$ en un seul schéma : $1W20^+$. ♣

Pour pouvoir manipuler les schémas et s'en servir pour prouver l'auto-stabilisation, il convient de définir une notion de réduction sur les schémas.

Définition 7.25. *Soit S un schéma, R une règle de réécriture et θ une substitution minimale. On dit que S se* réduit *en S' au moyen de la règle R et en utilisant la substitution θ, et on note $S\theta \underset{R}{\to} S'$ lorsque :*

$$S' = \{s' | \exists s \in S,\ s\theta \underset{R}{\to} s'\}.$$

Dans la pratique, ce ne sont pas les réductions décrites ci-dessus que l'on va considérer, mais plutôt une généralisation. En effet, si l'on considère le schéma $1W20^+$ de l'exemple précédent, il se réduit, *via* la règle M_4 et la substitution $\{W/W'0\}$ en $1W'200^+$. Si l'on considère la réduction sur les schémas définie ci-dessus, on va engendrer encore une infinité de schémas du type $1W20^j0^+$. Or c'est ce que sont censés éviter les schémas. L'idée de Beauquier *et al.* est donc de prendre une sur-approximation du S' défini ci-dessus. Ils considèrent donc la notion de *réduction généralisée* :

Définition 7.26. *On dit que le schéma T est obtenu à partir du schéma S par une* réduction généralisée *au moyen de la règle R et en utilisant la substitution θ, et on note $S\theta \nearrow_R T$, lorsqu'il existe un schéma S' tel que $S\theta \underset{R}{\to} S'$ et $S' \subseteq T$.*

De plus, dans les substitutions, par abus de notation, on ne va pas renommer les variables, c'est-à-dire que la substitution $\theta = \{W/W'0\}$ sera notée $\{W/W0\}$. Ainsi, en utilisant les réductions généralisées et sans renommer les variables on a $1W20^+\theta \nearrow_{M_4} 1W20^+$.

La notion de chaîne top s'étend de manière naturelle aux schémas, donnant des *chaînes top généralisées*. Seulement, comme on considère maintenant des réductions généralisées, le graphe associé à la relation de transition $\nearrow$ va être une sur-approximation du graphe de transition pour $\rightarrow$. Ainsi, s'il existe une dérivation cyclique dans le système original, elle va être détectée dans le graphe généralisé. Par contre une chaîne top généralisée quasi-cyclique ne correspond pas forcément à une véritable chaîne top quasi-cyclique.

En d'autre termes, si l'on prouve qu'il n'y a pas de cycle dans le système généralisé, on est sûr que le système original est auto-stabilisant mais si l'on détecte un cycle dans le système généralisé, il faut vérifier qu'il correspond bien à un cycle pour le système original.

Dans l'article [BBFM01], les auteurs donnent un algorithme pour construire les chaînes généralisées. Il faut cependant, à chaque étape, choisir le schéma $T \supseteq S'$ de manière non automatique et appropriée pour permettre de prouver l'auto-stabilisation.
Pour représenter ces chaînes, ils construisent un graphe généralisé associé à chaque règle de $Top_{\mathcal{S}}$, dont les sommets sont des schémas, et pour lequel une arête entre les sommets S et T étiquetée par R et θ signifie que $S\theta \nearrow_R T$. Un tel graphe est construit à l'aide d'une procédure, partant du membre droit de la règle de $Top_{\mathcal{S}}$, et ajoutant des nouveaux sommets au fur et à mesure du calcul, lorsqu'ils n'existent pas déjà.

Dans le cas où le système de réécriture vérifie la fermeture de $\mathcal{L}$ et l'absence de blocage hors de $\mathcal{L}$, la condition suffisante d'auto-stabilisation s'exprime de la façon suivante (voir [BBFM01] théorème 19, et sa preuve) :

Théorème 7.27. *Si, pour chaque règle $t \rightarrow u$ appartenant à $Top_{\mathcal{S}}$, dans le graphe généralisé associé, il n'y a pas de chemin qui utilise des règles de $Top_{\mathcal{S}}$ infiniment souvent (excepté les chemins qui arrivent dans $\mathcal{L}$), alors le système est auto-stabilisant.*

Comme annoncé plus haut, ce n'est qu'une condition suffisante, car l'existence de cycles hors de $\mathcal{L}$ n'empêche pas forcément que le système soit auto-stabilisant. Par contre, trouver un cycle qui n'existe pas dans le système réel peut permettre d'affiner la sur-approximation et ainsi, dans certains cas, de prouver l'auto-stabilisation.

7.7 L'outil de preuve Poulet

Pour mettre en œuvre cette méthode de vérification de systèmes de réécriture, Frédéric Magniette a créé un outil permettant de faciliter les preuves d'auto-stabilisation. Cet outil suit le principe de la méthode décrite précédemment : une partie de génération automatique du graphe exact, et une partie de généralisation manuelle.

L'outil peut donc engendrer pas à pas et de manière automatique les graphes de dérivation du premier ordre, et tenter de détecter des cycles dans ces graphes. Si l'auto-stabilisation ne peut pas directement être détectée, il est possible, au cours de la construction, d'observer le graphe construit et d'y introduire des schémas. L'outil détecte alors s'il y a des sommets à supprimer dans le graphe (par exemple des schémas qui forment un sous-ensemble du nouveau schéma que l'on a ajouté), et ajoute ou remplace le cas échéant certaines arêtes.
L'outil Poulet peut ensuite détecter des cycles éventuels dans le graphe fini ainsi obtenu. Lorsqu'il n'en existe pas, pour aucun des graphes de réduction associés aux règles de Top, alors on en déduit que le système est auto-stabilisant. Si l'algorithme détecte un cycle, il faut soit prouver que ce cycle correspond vraiment à un cycle dans le graphe exact, soit s'en servir pour restreindre les généralisations employées, afin de supprimer ces “faux” cycles.

Cet outil, ainsi qu'un guide utilisateur (en anglais) peut être trouvé sur internet à l'adresse : `http ://www.lri.fr/~magniett/poulet/index.html`.
Une description de l'outil, de son utilisation ainsi que des structures de données utilisées pour sa réalisation peut être également trouvée dans la thèse de Frédéric Magniette [Mag02].

7.8 Exemples

Pour illustrer cette méthode de preuve, l'article [BBFM01] présente trois exemples d'algorithmes auto-stabilisants, tirés des articles [BD95], [Gho93] et [Hoe94]. Les deux premiers sont présentés ici.

7.8.1 L'algorithme de Beauquier-Debas

Dans l'article [BD95], Beauquier et Debas présentent un algorithme auto-stabilisant d'exclusion mutuelle, qui est une adaptation de l'algorithme de Dijkstra à trois états [Dij74]. Le système total $\mathcal{S}$ peut se décrire de la manière suivante :

$$\begin{array}{llcll}
\textit{Bottom} & B_1 : 12X & \rightarrow & 21X & \\
\textit{Top} & T_1 : 0X0 & \rightarrow & 1X2 & \\
 & T_2 : 0X1 & \rightarrow & 1X0 & \\
 & T_3 : 0X2 & \rightarrow & 1X1 & \\
 & T_4 : 2X1 & \rightarrow & 1X2 & \\
 & T_5 : 2X2 & \rightarrow & 1X0 & \\
\textit{Middle} & M_1 : X10Y & \rightarrow & X01Y & (\text{ avec } X \neq \varepsilon) \\
 & M_2 : X11Y & \rightarrow & X02Y & (\text{ avec } X \neq \varepsilon) \\
 & M_3 : X12Y & \rightarrow & X00Y & (\text{ avec } X \neq \varepsilon) \\
 & M_4 : X02Y & \rightarrow & X20Y & (\text{ avec } X \neq \varepsilon) \\
 & M_5 : X22Y & \rightarrow & X10Y & (\text{ avec } X \neq \varepsilon)
\end{array}$$

L'ensemble légitime pour ce système est : $\mathcal{L} = 10^*20^* \cup 20^*10^*$.
Dans cet algorithme, l'état de départ du système n'est pas quelconque. En effet, du fait que les états des processus correspondent à la différence entre les états des processus de l'algorithme de Dijkstra à trois états, la somme de tous les états des processus dans l'algorithme présenté ci-dessus doit être nulle modulo 3. Il est facile de voir que tout mot clos vérifiant la condition précédente est réductible pour ce système, et que toute réduction préserve la somme des valeurs des états. De plus, l'ensemble $\mathcal{L}$ est fermé pour $\mathcal{S}$. Il suffit donc de montrer qu'il n'y a pas de dérivation close cyclique pour $\mathcal{S}$ contenant un mot non légitime.

On constate que les règles T_1, T_2 et T_3 sont appliquées au plus une fois le long d'une exécution. En effet, dans le membre gauche de chacune de ces règles, il y a un 0 en position bottom mais plus dans le membre droit, et aucune autre règle ne produit de 0 à cette position (en particulier les règles de *Middle* qui ne peuvent pas y être appliquées). Il suffit donc de considérer les dérivations de $\mathcal{S}/\{T_1, T_2, T_3\}$ et de vérifier qu'elles n'ont pas de cycle hors de $\mathcal{L}$.

Ensuite, Beauquier *et al.* considèrent, afin de simplifier encore le système, la mesure $\varphi_{\mathcal{BD}}$ qui compte le nombre de processus qui sont dans un état différent de 0. Une simple observation des règles montre que $\varphi_{\mathcal{BD}}$ ne croît jamais lorsqu'on applique une règle de $\mathcal{S}/\{T_1, T_2, T_3\}$, et on va pouvoir, d'après le théorème 7.21, ne considérer que le système $\mathcal{BD}$ présenté dans l'exemple 7.2, page 121.

$$\begin{array}{ll} B_1: & 12X \to 21X \\ M_1: & X10Y \to X01Y \ \text{(avec } X \neq \varepsilon\text{)} \\ M_4: & X02Y \to X20Y \ \text{(avec } X \neq \varepsilon\text{)} \\ T_4: & 2X1 \to 1X2 \end{array}$$

Ici, trouver un système correspondant qui préserve $\varphi_{\mathcal{BD}}$ consiste juste à enlever des règles. Cependant, il arrive que ce soit plus compliqué, comme on le verra pour l'algorithme de Ghosh à la section 7.8.2.

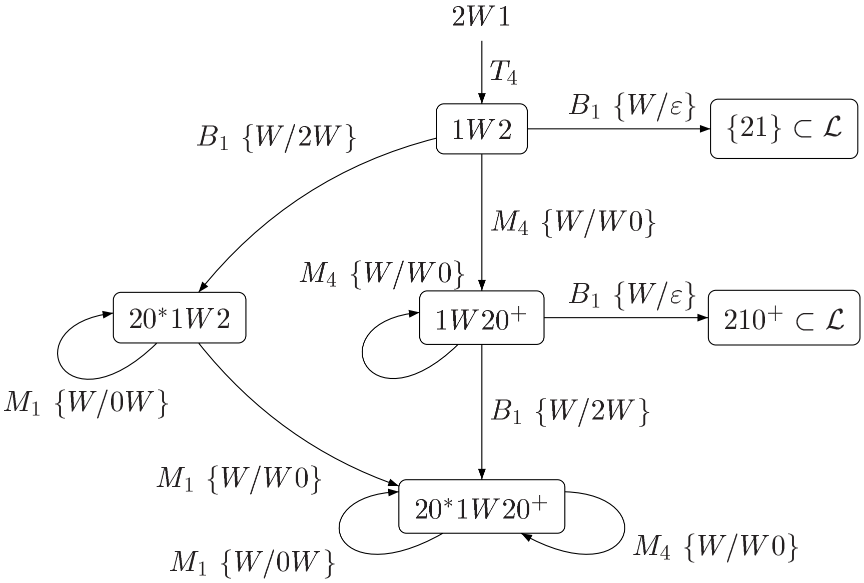

FIG. 7.1 – Graphe généralisé associé à l'algorithme de Beauquier-Debas

Comme on l'a vu à l'exemple 7.9, page 123, le système $\mathcal{BD} - T_4$ est nœtherien. Or on constate dans le graphe généralisé, construit dans l'article [BBFM01] et présenté à la figure 7.1 qu'il n'existe pas de cycle hors de $\mathcal{L}$ qui contienne une application de T_4. Ceci permet de conclure à l'auto-stabilisation de l'algorithme de Beauquier-Debas.

7.8.2 L'algorithme de Ghosh

Cet algorithme, présenté dans l'article [Gho93], est pour sa part une variante de l'algorithme de Dijkstra à quatre états. Le système se compose d'une chaîne de N machines (où, rappelons le, N est un paramètre), numérotées de 0 à $N-1$, mais dans ce cas ci les machines d'indice 0 (bottom) et $N-1$ (top) ne sont pas reliées entre elles. L'espace d'états possible pour les différents processus est $\Sigma = \{0, 1, 2, 3\}$, sauf pour le processus bottom dont l'espace d'états est $\{1, 3\}$, et le processus top dont l'espace d'états est $\{0, 2\}$. Comme dans l'exemple précédent, l'ensemble des configurations dont on part est quelque peu restreint, du fait de la condition ci-dessus, et l'application d'une règle nous fait rester dans cet ensemble de configurations.

Le système de réécriture correspondant à l'algorithme de Ghosh est le suivant :

$$\begin{array}{llccc}
\textit{Middle} & F_1(q): & Xq(q+1)Y & \to & X(q+1)(q+1)Y \\
 & F_2(q): & X(q+1)qY & \to & X(q+1)(q+1)Y \\
\textit{Top} & E_1: & X32 & \to & X30 \\
 & E_2: & X10 & \to & X12 \\
\textit{Bottom} & C_1: & 12X & \to & 32X \\
 & C_2: & 30X & \to & 10X
\end{array}$$

où q représente un état quelconque de Σ, et l'addition est effectuée modulo 4. Comme on est dans le cas d'une topologie linéaire, et qu'on ne veut pas confondre les règles de *Middle* (qui ne modifient pas les processus des extrémités) avec celles de *Top* et *Bottom*, on va supposer que les variables X dans F_2 et Y dans F_1 ne représentent pas le mot vide. Cela permet d'assurer qu'elles ne modifient pas les processus top et bottom.

Étant donnée une configuration $t = a_0...a_{N-1}$, on dit que le processus d'indice i possède un *privilège* si son état est égal à celui de l'un de ses voisins moins 1, c'est-à-dire $a_i = a_{i-1} - 1$ ou $a_i = a_{i+1} - 1$ lorsque ces voisins existent. Les processus ayant des privilèges sont donc ici ceux auxquels on peut appliquer une règle. L'ensemble légitime est $\mathcal{L} = \{1^+, 3^+\}\{0^+2^+\}$, c'est-à-dire l'ensemble des configurations à exactement un privilège.

Étant donnée une configuration $t = a_0...a_{N-1}$, on dit qu'il y a un *break* entre les positions i et $i+1$ ($i < N-1$)lorsque $a_i \neq a_{i+1}$. La preuve originale de Ghosh utilise une fonction $f = (B, D)$ sur les configurations, dont la première composante B compte le nombre de *breaks*, et la deuxième composante D compte la somme des distances entre les paires de breaks consécutifs.

L'idée de Beauquier *et al.* est de ne considérer que la fonction B, qui ne croît jamais lors d'une réduction, et de la prendre comme fonction $\varphi_{\mathcal{G}}$. Il est possible de construire un système de réécriture correspondant préservant $\varphi_{\mathcal{G}}$. Les règles de *Top* et *Bottom* sont conservées, mais par contre il faut changer les règles de *Middle* de la façon suivante.

$$\begin{array}{llccc}
\textit{Middle} & D_1(q): & Xqq(q+1)Y & \to & Xq(q+1)(q+1)Y \\
 & D_2(q): & X(q+1)qqY & \to & X(q+1)(q+1)qY
\end{array}$$

Ainsi on est sûr que le break va se déplacer d'une lettre vers la droite (resp. la gauche) mais aucun ne va disparaître. On remarque que cette modification introduit des règles à trois lettres, mais qui ne modifient que la lettre centrale. Comme on est passé de règles à deux lettres aux règles à trois lettres (qui ne modifient que la lettre centrale), on n'a plus besoin de supposer que les variables X et Y ne représentent pas le mot vide.

Il faut encore vérifier que tout mot est réductible pour le système original, et que $\mathcal{G} - Top_{\mathcal{G}}$ est nœtherien. Tout mot est bien réductible car, les deux processus des extrémités étant de parité différente, il existe, dans toute configuration, au moins deux processus consécutifs dont les états q_i et q_{i+1} sont de parité différente. En raisonnant modulo 4 on a forcément $q_i = q_{i+1}+1$ (cas 1) ou $q_{i+1} = q_i+1$ (cas 2). Si on est dans le cas 1, et si $0 \leq i < N-2$ on peut appliquer une règle du système original au processus d'indice $i+1$. Si on est dans le cas 2 et $1 \leq i \leq N-2$ on peut appliquer une règle du système original au processus d'indice i. Si on est dans l'un des deux cas restants, on peut appliquer soit une règle de *Top* soit une de *Bottom* selon le cas. Pour ce qui est de l'aspect nœtherien, supposons qu'on enlève les deux règles de $Top_{\mathcal{G}}$. Alors le processus d'indice $N-1$ ne sera jamais réécrit, et gardera toujours la même valeur, mettons q. Le processus d'indice $N-2$ peut alors être réécrit au plus deux fois : $(q-1)(q-1)q \xrightarrow[D_1(q-1)]{} (q-1)qq$ puis, après d'autres réductions $(q+1)qq \xrightarrow[D_2(q)]{} (q+1)(q+1)q$. Ensuite il sera impossible de réécrire le processus $N-2$ sans réécrire le processus $N-1$, car il n'existe aucun membre gauche de règle du genre $q'(q+1)q$. Ainsi le processus d'indice $N-2$ ne va plus être réécrit à partir d'un certain moment. En itérant ce raisonnement, on obtient que toute dérivation qui ne réécrit pas le processus le plus à droite est finie.

Le graphe généralisé correspondant à cet algorithme est présenté dans [BBFM01]. Il montre qu'il n'existe pas de chaîne top quasi-cyclique, et donc pas de cycle qui ne soit pas dans $\mathcal{L}$. Ce système est donc auto-stabilisant vers $\mathcal{L}$, l'ensemble des configurations à exactement un privilège.

Chapitre 8

Preuve automatisée de convergence par réécriture préfixe

La plupart des résultats de ce chapitre a fait l'objet d'une publication avec Laurent Fribourg et Ulf Nilsson, dans l'article [DFN01].

Dans ce chapitre, nous allons présenter une amélioration de la méthode décrite au chapitre précédent. En partant des travaux de Beauquier *et al*, nous avons effectué une étape de plus vers l'automatisation de telles preuves de convergence. Au lieu de construire plus ou moins manuellement des graphes généralisés et de tester s'ils contiennent des cycles, nous calculons automatiquement un langage régulier qui, sous certaines conditions, sera inévitable (voir définition 3.12, page 57).
En considérant les étapes de surréduction comme de la réécriture préfixe et suffixe, nous avons utilisé un résultat dû à Caucal, annonçant que le langage engendré par réécriture suffixe (resp. préfixe) est régulier et peut se calculer en temps polynomial. L'article [Cau88] donne également un algorithme de construction de l'automate correspondant.
Nous avons ensuite exhibé une condition suffisante, vérifiable directement sur les règles, assurant que l'ensemble $\mathcal{N_S}^*$ (obtenu à partir des membres droits de règles de $Top_{\mathcal{S}}$ en appliquant des surréductions itérées) est fermé pour $\mathcal{S}$, puis implanté l'algorithme de Caucal dans un programme Prolog, que nous avons testé sur différents exemples.

La section 8.1 présente tout d'abord les résultats de réécriture préfixe que nous allons utiliser. Les surréductions sont alors à nouveau présentées, dans ce contexte, et reliées à la réécriture préfixe dans la section 8.2. Nous donnons ensuite, aux sections 8.3 et 8.4, un critère permettant de prouver que l'ensemble d'états $\mathcal{L}$ est *absorbant*, puis une condition facilement vérifiable permettant d'appliquer notre méthode .

8.1 Quelques résultats de réécriture préfixe

Avant de montrer en quoi elle permet d'engendrer des ensembles réguliers et en quoi elle peut nous servir, il faut tout d'abord définir précisément ce qu'est la *réécriture préfixe* (resp.*suffixe*).

Définition 8.1. *Étant donné un système de réécriture $\mathcal{S}$ fini, on dit qu'un mot clos t_2 est obtenu par* réduction préfixe *(resp.* réduction suffixe*) d'un mot clos t_1 s'il existe une règle $R : u \to v$ de $\mathcal{S}$ et des facteurs t'_1 et t'_2 tels que $t_1 = ut'_1$ et $t_2 = vt'_2$ (resp. $t_1 = t'_1u$ et $t_2 = t'_2v$).*

Ainsi, la réécriture préfixe consiste à appliquer les règles en début de mot, et la réécriture suffixe en fin de mot.

Dans l'article [Cau90] (voir aussi [Cau88] pour la version avec preuve), Caucal énonce (et prouve) le théorème suivant :

Théorème 8.2. *Pour tout système de réécriture $\mathcal{S}$, la relation $\underset{\mathcal{S}}{\rightarrow}^*$ correspondant à la réécriture préfixe itérée est une transduction rationnelle, et le transducteur associé est constructible en pratique à partir de $\mathcal{S}$.*

Pour des définitions de transducteur et transduction rationnelle, on peut regarder par exemple [AU72].

De plus, Caucal donne un algorithme en temps polynomial pour construire le transducteur ci-dessus. La proposition (tirée de [Cau88]) qui va nous intéresser est quelque peu plus restreinte. En fait elle traite le cas de la réduction suffixe, mais la preuve pour la réduction préfixe est similaire :

Proposition 8.3. *Pour tout mot t appartenant à Σ^*, le langage des mots accessibles par réduction suffixe à partir de t est régulier, et on peut construire en temps polynomial un automate fini le reconnaissant.*

Le premier résultat annonçant que le langage engendré par des réductions préfixes (ou suffixes) itérées est régulier est dû à Büchi (voir [Büc89] chapitre 5). Il affirme également que l'automate reconnaissant ce langage est effectivement constructible mais ne donne pas de méthode polynomiale. D'autres travaux ont été fait sur le sujet, mais à notre connaissance aucun, avant Caucal, n'a donné de construction polynomiale.

Nous n'allons pas donner la preuve de la proposition 8.3, mais plutôt une description de la construction de l'automate voulu.

Soit $\mathcal{S}$ un système de réécriture. Définissons tout d'abord l'ensemble $MbG(\mathcal{S})$, composé de tous les membres gauches de règles de $\mathcal{S}$, à savoir

$$MbG(\mathcal{S}) = \{u \in \Sigma^+ | \exists v \in \Sigma^+ : \ u \to v \text{ est une règle de } \mathcal{S}\}$$

et $E = \{\varepsilon\} \cup MbG(\mathcal{S})$ qui va désigner l'ensemble des états que nous allons considérer.
Pour donner une idée de la construction, il faut considérer des systèmes de transitions étiquetés. Nous allons considérer des transitions qui ne sont plus des éléments de $E \times E$, mais des éléments de $E \times \Sigma^* \times E$, la deuxième composante étant un mot appelé *étiquette* de la transition.
La composition se définit alors sur les systèmes de transitions étiquetés finis par :

$$A \circ B = \{(s, uv, t) \,|\, \exists r : \ (s, u, r) \in A \text{ et } (r, v, t) \in B\}.$$

Nous pouvons ainsi définir la fermeture réflexive transitive de A pour la composition $\circ$, que nous noterons A^*.

Ensuite, étant donné un système de transitions fini étiqueté A , on va considérer son *découpage* $\langle A \rangle$, défini comme suit :

$$\langle A \rangle = \{(s, u', u''v) \notin A^* \,|\, \exists u, y : \ (s, u, y) \in A \wedge (y, v, \varepsilon) \in A^* \wedge (u'' \text{ plus grand suffixe de } u \text{ tel que } u'' \neq \varepsilon \text{ et } u''v \in MbG(\mathcal{S})) \wedge u'u'' = u\}$$

Exemple 8.4. Considérons le système de réécriture $\mathcal{S} = \{(\varepsilon, ab), (bab, ab)\}$. Les règles sont ici représentées sous forme de couples. Supposons que le système de transitions A se compose des deux seules transitions (ba, ab, b) et (b, ab, ε). Le découpage du système A se fait alors comme représenté à la figure 8.1.
Ici $s = ba$, $u = ab$, $y = b$ et $v = ab$. Le plus long préfixe non vide u'' de u tel que $u''v \in MbG(\mathcal{S})$ est donc b. La nouvelle transition obtenue par découpage est $(s, u', u''v) = (ba, a, bab)$.

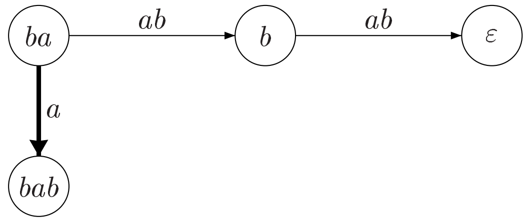

FIG. 8.1 – Exemple de découpage

La flèche représentée en gras désigne la transition ajoutée lors du découpage. ♣

On définit alors le complété d'un système de transitions étiqueté A par :

$$\overline{A} = \begin{cases} A \text{ si } \langle A \rangle = \emptyset \\ \overline{A \cup \langle A \rangle} \text{ sinon.} \end{cases}$$

De plus, pour tout mot $u \in \Sigma^*$, on note $Suf(u)$ le plus grand suffixe de u appartenant à E et $Pref(u)$ le préfixe de u tel que $u = (Pref(u).Suf(u))$. Ceci nous permet de définir le graphe étiqueté $G(\mathcal{S})$ associé au système de réécriture $\mathcal{S}$:

$$\begin{array}{rcl} H_{\mathcal{S}} & = & \{(x, Pref(y), Suf(y)) \mid x \neq y \text{ et } x \to y \in \mathcal{S}\} \\ I_{\mathcal{S}} & = & \{(ay, a.Pref(y), Suf(y)) \mid ay \in MbG(\mathcal{S}) \text{ et } a \in \Sigma\} \\ G(\mathcal{S}) & = & \overline{H_{\mathcal{S}} \cup I_{\mathcal{S}}} \end{array}$$

À partir de cette construction, on obtient la proposition suivante, toujours tirée de [Cau88] :

Proposition 8.5. *Pour tout mot $u \in \Sigma^*$, le langage engendré par réécriture suffixe itérée à partir de u est reconnu par l'automate $\mathcal{A}_{\mathcal{S},u}$ dont les transitions et les états sont ceux de $G(S \cup \{(u, u)\})$, l'état initial est celui étiqueté par u, et l'état final est celui étiqueté par ε.*

En fait, il faut parfois raffiner cette construction, car une transition $s \xrightarrow{ab} s'$ dans $G(S \cup \{(u, u)\})$ va être transformée en deux transitions $s \xrightarrow{a} s''$ et $s'' \xrightarrow{b} s'$, où s'' est un nouvel état créé lors de la transformation en automate.

Ainsi, le langage des mots engendrés par réécriture suffixe à partir de u est composé de toutes les étiquettes des chemins de $G(\mathcal{S} \cup \{(u, u)\})$ partant de l'état u et arrivant à l'état ε. Pour la preuve, se référer à l'article.

Exemple 8.6. Pour terminer cette section, voici un exemple permettant d'illustrer la méthode. Considérons à nouveau le système $\mathcal{S} = \{(\varepsilon, ab), (bab, ab)\}$ de l'exemple précédent, et choisissons le mot $u = ab$. Le système à considérer pour construire G est $\mathcal{S}' = \mathcal{S} \cup \{(u,u)\}$. On a :

$$\begin{array}{rcl} H_{\mathcal{S}'} & = & \{(\varepsilon, ab, \varepsilon), (bab, ab, \varepsilon)\} \\ I_{\mathcal{S}'} & = & \{(bab, b, ab), (ab, ab, \varepsilon)\} \\ \langle H_{\mathcal{S}'} \cup I_{\mathcal{S}'} \rangle & = & \{(bab, a, bab), (ab, a, bab), (\varepsilon, a, bab)\} \end{array}$$

De plus $\langle H_{\mathcal{S}'} \cup I_{\mathcal{S}'} \cup \langle H_{\mathcal{S}'} \cup I_{\mathcal{S}'} \rangle \rangle$ est vide, et donc $\overline{H'_{\mathcal{S}} \cup I_{\mathcal{S}'}} = \langle H_{\mathcal{S}'} \cup I_{\mathcal{S}'} \cup \langle H_{\mathcal{S}'} \cup I_{\mathcal{S}'} \rangle \rangle$. Le graphe $G(\mathcal{S}')$ correspondant est présenté à la figure 8.2. Pour obtenir un automate fini, il suffit alors de rendre final l'état ε, initial l'état ab et de remplacer les transitions étiquetées par deux lettres en deux transitions étiquetée chacune par une lettre, avec un nouvel état intermédiaire.

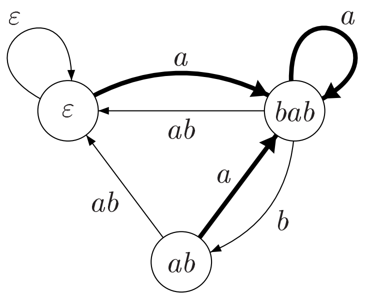

Fig. 8.2 – Le système de transitions étiqueté $G(\mathcal{S}')$.

Comme précédemment, les flèches représentées en gras sont celles ayant été ajoutées lors du découpage.
Le langage accepté par l'automate $\mathcal{A}_{\mathcal{S},u}$ peut donc être calculé, et vaut $(a^+b)^*$. ♣

Comme on le verra à la section 9.3, nous avons implanté cet algorithme en Prolog pour effectuer des preuves de convergence.

8.2 Retour sur les surréductions

Comme il a été dit dans l'introduction de ce chapitre, nous allons utiliser une méthode s'appuyant sur de la réécriture préfixe. C'est pourquoi cette section va entre autres servir à faire le lien entre la notion de surréduction et celle de réécriture préfixe/suffixe.

Comme cela avait été fait pour les schémas au chapitre précédent, on peut décrire la réduction d'un ensemble de mots. On définit $Reduce_{\mathcal{S}}(J) = \{t' \mid \exists t \in J : \ t \rightarrow_{\mathcal{S}} t'\}$ qui est l'ensemble des mots qui peuvent être obtenus à partir d'un mot de J en une réduction *via* $\mathcal{S}$. L'ensemble J est dit *fermé* pour le système de réécriture $\mathcal{S}$ si $Reduce_{\mathcal{S}}(J) \subseteq J$.

La fonction $Reduce_{\mathcal{S}}$ peut être vue comme un *transducteur* (voir par exemple [KMM+97]) et si J est régulier, alors $Reduce_{\mathcal{S}}(J)$ est régulier.

Comme on a un système paramétré, on sait d'après [AK86] que l'ensemble obtenu à partir de J en appliquant $\underset{\mathcal{S}}{\rightarrow}^*$ n'est pas régulier dans le cas général, même lorsque l'ensemble de départ J est régulier. Cependant, comme on l'a vu à la section 8.1, l'application itérée de réductions *préfixes* (appliquées au début du mot) et *suffixes* (appliquées à la fin du mot) peut être représentée par un transducteur et l'automate correspondant au langage $Narrow^*_{\mathcal{S}}(J)$ peut être construit en temps polynomial par rapport à la taille du système $\mathcal{S}$.

Dans ce chapitre, nous allons utiliser des stratégies de réécriture préfixe et suffixe afin de pouvoir appliquer les résultats de Caucal, présentés à la section 8.1.

On peut tout d'abord noter que tester la fermeture d'un ensemble régulier J pour $Reduce_{\mathcal{S}}$ est décidable : cela consiste à appliquer un transducteur puis tester l'inclusion de deux langages réguliers.

8.2.1 Extension de règles

Pour décrire les stratégies de réécriture préfixe et suffixe, il convient de définir pour chaque règle M de $Middle_{\mathcal{S}}$ (resp. pour chaque règle B de $Bottom_{\mathcal{S}}$) deux variantes de M (resp. une variante de B) comme suit :

Définition 8.7. *Étant données une règle M : $XabY \to Xa'b'Y$ de $Middle_{\mathcal{S}}$ et une règle B : $abY \to a'b'Y$ de $Bottom_{\mathcal{S}}$,*

- *l'*extension préfixe *de M, notée M_{pre}, est : $X\odot bY \to X\odot a'b'Y$,*
- *l'*extension suffixe *de M, notée M_{suf}, est : $Xa\odot Y \to Xa'b'\odot Y$,*
- *l'*extension suffixe *de B, notée B_{suf}, est : $a\odot Y \to a'b'\odot Y$,*

où $\odot$ est une nouvelle constante ajoutée à l'alphabet Σ.

Les extensions présentées ci-dessus vont servir à simuler les opérations de surréduction à droite (pour M_{pre}), et à gauche (pour M_{suf} et B_{suf}), comme on le verra dans la suite de cette section. Comme ces règles correspondent à une substitution suivie de l'application d'une règle de $\mathcal{S}$, elles ne préservent pas la longueur.
Cette définition est présentée pour des règles à deux lettres, mais elle peut s'étendre pour des règles à trois lettres. Dans le cas de règles à 3 lettres, en général, il faut également considérer des extensions préfixes et suffixes correspondant à une substitution à deux lettres. Par exemple $X\odot cY \to X\odot a'b'c'Y$ est une extension préfixe de la règle $XabcY \to Xa'b'c'Y$ de $Middle_{\mathcal{S}}$. Cependant, pour l'algorithme à trois lettres que nous allons considérer ([Gho93]), il n'est pas nécessaire de considérer ces substitutions à deux lettres car les règles ne modifient que la lettre centrale (voir section 7.8.2). Ainsi, en utilisant le résultat de Dershowitz dont nous avons déjà parlé à la section 7.4, nous pouvons ne considérer que des substitutions à une lettre car ce sont les seules qui réécrivent une lettre de la partie active (cf. section 7.4).

Soit $t = t_1Wt_2$ un mot du premier ordre. Notons $\tilde{t}$ le mot clos $t_1\odot t_2$ obtenu en remplaçant dans t la variable W par la constante $\odot$.

Si le facteur t_2 commence par la lettre b, on peut appliquer une réduction *via* la règle M_{pre} à $\tilde{t}$. Une telle réduction affecte un préfixe du facteur $\odot t_2$ et est donc appelée *réduction préfixe* de la partie droite de $\tilde{t}$. De la même manière, une réduction *via* M_{suf} (resp. B_{suf}) affecte un suffixe de $t_1\odot$ et est appelée *réduction suffixe* de la partie gauche de $\tilde{t}$.

Ces extensions de règles vont nous permettre de considérer les opérations de surréduction définies à la section 7.3.2 comme une forme de réduction préfixe ou suffixe, et ainsi d'appliquer l'algorithme de Caucal.
Ainsi, à la surréduction à droite $t \rightsquigarrow_M^{droite} t'$ correspond une réduction préfixe de $\tilde{t}$ en $\tilde{t'}$ au moyen de la règle M_{pre}. En effet on a :

$$\underset{t}{\underset{\shortparallel}{uWbv}} \rightsquigarrow_M^{droite} \underset{t'}{\underset{\shortparallel}{uWa'b'v}} \iff \underset{\tilde{t}}{\underset{\shortparallel}{u\odot bv}} \to_{M_{pre}} \underset{\tilde{t'}}{\underset{\shortparallel}{u\odot a'b'v}}.$$

De la même manière, à la surréduction à gauche $t \rightsquigarrow_M^{gauche} t'$ (resp. la surréduction “bottom” $t \rightsquigarrow_B^{bot} t'$) correspond une réduction suffixe de $\tilde{t}$ en $\tilde{t'}$ en utilisant la règle M_{suf} (resp. B_{suf}). On a également l'équivalence :

$$uaWv \rightsquigarrow_M^{gauche} ua'b'Wv \Longleftrightarrow ua\odot v \rightarrow_{M_{suf}} ua'b'\odot v$$

$$(\text{resp. } aWv \rightsquigarrow_B^{bot} a'b'Wv \Longleftrightarrow a\odot v \rightarrow_{B_{suf}} a'b'\odot v).$$

Exemple 8.8. En utilisant la règle $M_4 : X02Y \rightarrow X20Y$ du système $\mathcal{BD}$ (voir exemple 7.2, page 121), on obtient la réduction $01W200 \rightsquigarrow_{M_4}^{droite} 01W2000$. Cette surréduction est simulée par la réduction $01\odot 200 \rightarrow_{M_{4pre}} 01\odot 2000$ en utilisant l'extension préfixe $M_{4pre} :\ X\odot 2Y \rightarrow X\odot 20Y$. ♣

8.2.2 Ensembles engendrés par réductions et surréductions

Étant donné un ensemble J de mots du premier ordre, en utilisant la définition de la fonction $Reduce_{\mathcal{S}}$ donnée à la section 8.2 on note :

$$\begin{array}{rcl} Narrow_{\mathcal{S}}(J) & = & \{t' \mid \exists t \in J :\ t \rightsquigarrow_{\mathcal{S}} t'\} \\ Narrow^*_{\mathcal{S}}(J) & = & \{t' \mid \exists t \in J :\ t \rightsquigarrow^*_{\mathcal{S}} t'\} \\ (Narrow_{\mathcal{S}} + Reduce_{\mathcal{S}})^*(J) & = & \{t' \mid \exists t \in J :\ t\ (\rightsquigarrow_{\mathcal{S}} \cup \underset{\mathcal{S}}{\rightarrow})^*\ t'\}. \end{array}$$

Les deux premiers ensembles sont obtenus en appliquant (de façon répétée pour le deuxième) l'opération de surréduction. Le troisième est le résultat de réductions et de surréductions entremêlées.

On appelle *ensemble initial* de $\mathcal{S}$ et on note $I_{\mathcal{S}}$ l'ensemble constitué des membres droits de règles de $Top_{\mathcal{S}}$, à savoir des mots du type $b'Wa'$ dans le cas de la topologie en anneau, et $Wa'b'$ dans le cas de la topologie linéaire[1].

Remarque : L'ensemble $I_{\mathcal{S}}$ est appelé ensemble initial, non pas parce que toutes les dérivations commencent par une configuration de $I_{\mathcal{S}}$, mais parce que cet ensemble est le point de départ des chaînes de surréductions que nous allons étudier.

Nous allons maintenant considérer des ensembles engendrés par surréductions et/ou réductions, mais cette fois-ci en partant précisément de l'ensemble initial $I_{\mathcal{S}}$. On note :

$$\begin{array}{rcl} \mathcal{N}^*_{\mathcal{S}} & = & Narrow^*_{\mathcal{S}}(I_{\mathcal{S}}) \\ (\mathcal{N}_{\mathcal{S}} + \mathcal{R}_{\mathcal{S}})^* & = & (Narrow_{\mathcal{S}} + Reduce_{\mathcal{S}})^*(I_{\mathcal{S}}). \end{array}$$

Dans le calcul de $\mathcal{N}^*_{\mathcal{S}}$, les surréductions à droite d'une part, et les surréductions à gauche d'autre part sont effectuées de manière indépendante. En effet, elles concernent deux facteurs séparés par la variable W. Cette indépendance permet de les effectuer en parallèle. Étant donné que les surréductions à droite (resp. les surréductions à gauche) peuvent être simulées par des étapes de réduction préfixe (resp. réduction suffixe), $\mathcal{N}^*_{\mathcal{S}}$ peut être engendré en appliquant en parallèle des étapes de réduction préfixe d'une part, et de réduction suffixe d'autre part, en partant des membres droits de règles de $Top_{\mathcal{S}}$. En appliquant le résultat de Caucal décrit à la section 8.1, on a directement :

[1] La variable X a été remplacée par W car on représente des configurations et non des règles.

Proposition 8.9. *L'ensemble $\mathcal{N}_{\mathcal{S}}^*$ est régulier, et peut être construit en temps polynomial en la taille de $\mathcal{S}$.*

On décompose l'ensemble $\mathcal{N}_{\mathcal{S}}^*$ en $\mathcal{M}_0 \cup \mathcal{M}_1$, suivant le nombre de fois où l'on a appliqué une surréduction associée à *Bottom*. $\mathcal{M}_0$ (resp. $\mathcal{M}_1$) désigne le sous-ensemble engendré avec zéro (resp. une) application de surréduction avec une règle de $Bottom_{\mathcal{S}}$ [2]. Plus précisément :
- $\mathcal{M}_0$ est obtenu en appliquant $(\rightsquigarrow^{droite})^*$ à $I_{\mathcal{S}}$ (on ne peut pas réduire la variable à gauche sans avoir d'abord appliqué une surréduction *via* une règle de $Bottom_{\mathcal{S}}$), et
- $\mathcal{M}_1$ est obtenu en appliquant à $I_{\mathcal{S}}$ des surréductions du type $(\rightsquigarrow^{gauche})^* \circ (\rightsquigarrow^{bot})$ et $(\rightsquigarrow^{droite})^*$ en parallèle.

Exemple 8.10. Si l'on reprend le système $\mathcal{BD}$ de l'exemple 7.2 page 121, version simplifiée de l'algorithme de Beauquier-Debas, il n'y a qu'une règle dans $Top_{\mathcal{BD}}$. Ainsi l'ensemble initial $I_{\mathcal{BD}}$ est $1W2$.
L'application de $(\rightsquigarrow^{droite})^*$ à l'ensemble initial $I_{\mathcal{BD}}$ au moyen de la règle $M_4 : X02Y \to X20Y$ (qui est la seule applicable à droite) donne $\{1W2, 1W20, 1W200, 1W2000, \dots\}$. Ainsi l'ensemble $\mathcal{M}_0$ est $1W20^*$.
D'autre part, l'application de $\rightsquigarrow^{bot}$ à $I_{\mathcal{BD}}$ au moyen de la règle B_1 donne $21W2$. Ensuite, l'application de $(\rightsquigarrow^{gauche})^*$ au moyen de la règle $M_1 : X10Y \to X01Y$ (seule applicable à gauche) donne $\{21W2, 201W2, 2001W2, \dots\}$, c'est-à-dire 20^*1W2. En appliquant en parallèle $(\rightsquigarrow^{droite})^*$, on obtient finalement l'ensemble $\mathcal{M}_1 = 20^*1W20^*$. L'ensemble engendré par surréductions vaut alors $\mathcal{N}_{BD}^* = \mathcal{M}_0 \cup \mathcal{M}_1 = 1W20^* \cup 20^*1W20^*$. ♣

8.2.3 Surréductions closes

La notion de surréduction close va être importante pour notre étude car, contrairement à l'article [BBFM01], nous n'allons considérer que des dérivations issues de top qui contiennent une surréduction close, c'est-à-dire dans lesquelles on va finir par faire disparaître la variable.

Considérons à présent un peu plus en détail l'opération de surréduction close. Dans le cas des règles à deux lettres, elle va tout simplement consister à remplacer la variable W par ε, puis appliquer une réduction, obtenant ainsi un mot clos.
En effet, si on se rappelle de la définition des substitutions minimales donnée à la section 7.3.1, on avait des substitutions closes de la forme $\{W/a_{i+1}...a_j\}$ avec $1 \leq i \leq j \leq n-1$ où n est le nombre de lettres de la règle considérée. Ici $n = 2$, et donc $i = j$ et la variable W est obligatoirement changée en ε. Dans le cas des règles à trois lettres, on peut envisager une substitution minimale de la forme $\{W/b\}$. Dans le cas de l'algorithme $\mathcal{G}$ avec des règles à trois lettres, et parce que ces règles ne modifient que la lettre centrale, on n'aura besoin que d'instanciations closes du type $\{W/\varepsilon\}$ (toujours d'après le résultat de Dershowitz vu à la section 7.4).

La définition suivante présente les deux cas où peut avoir lieu une surréduction close, et ce lorsque l'on a des règles à deux lettres. L'extension de cette définition pour des règles à trois lettres est assez naturelle.

Définition 8.11. *Considérons les deux règles suivantes : $M : XabY \to Xa'b'Y$ appartenant à $Middle_{\mathcal{S}}$ et $B : abY \to a'b'Y$ appartenant à $Bottom_{\mathcal{S}}$. Soit t un mot du premier ordre et t' un mot clos.*

[2] Il est facile de voir que, comme toutes les règles de *Bottom* concernent le même nombre de lettres, on applique au plus une fois une surréduction utilisant une règle de $Bottom_{\mathcal{S}}$.

- *On dit qu'une* surréduction close *s'applique à t pour donner le mot t' au moyen de la règle M s'il existe des facteurs clos $u \in \Sigma^+$ et $v \in \Sigma^*$ tels que $t = uaWbv$ et $t' = ua'b'v$. Cette surréduction est notée $t \rightsquigarrow_M^{clos} t'$;*
- *on dit qu'une* surréduction close *s'applique à t pour donner le mot t' au moyen de la règle B s'il existe un facteur $v \in \Sigma^*$ tel que $t = aWbv$ et $t' = a'b'v$. Cette surréduction est notée $t \rightsquigarrow_B^{clos} t'$.*

Étant donné un ensemble J de mots du premier ordre, l'image de J par application d'une surréduction close *via* $\mathcal{S}$, notée $Gr_{\mathcal{S}}(J)$ ou plus simplement $Gr(J)$ est

$$Gr(J) = \{t' \mid \exists t \in J : \ t \rightsquigarrow_{\mathcal{S}}^{clos} t'\}.$$

On définit également l'ensemble $\mathcal{G}_{\mathcal{S}} = Gr(\mathcal{N}_{\mathcal{S}}^*)$, obtenu à partir de $I_{\mathcal{S}}$ en appliquant un nombre fini de fois des surréductions non closes puis une surréduction close.

La fonction Gr peut être vue comme un transducteur. Ainsi, si l'ensemble J est régulier, $Gr(J)$ l'est aussi. En particulier, en utilisant le résultat de Caucal, comme $I_{\mathcal{S}}$ est régulier car fini, $\mathcal{G}_{\mathcal{S}}$ est régulier. Il peut être obtenu en appliquant une surréduction close à l'aide d'une règle de *Middle* à $\mathcal{M}_1$, et à l'aide d'une règle de *Bottom* à $\mathcal{M}_0$.

Exemple 8.12. Toujours avec le système $\mathcal{BD}$, et ses ensembles $\mathcal{M}_0$ et $\mathcal{M}_1$ qui ont été calculés à l'exemple 8.10, page 141, l'application d'une surréduction close à l'ensemble $\mathcal{M}_0 = 1W20^*$ donne 210^*. La règle employée est $B_1 \ : \ 12X \to 21X$. Il n'y a pas de surréduction close applicable aux mots de l'ensemble $\mathcal{M}_1 = 20^*1W20^*$, que ce soit pour la règle M_1 ou M_4. On a donc $\mathcal{G}_{\mathcal{BD}} = Gr(\mathcal{M}_0) = 210^*$. ♣

8.3 Langages réguliers inévitables et convergence

Dans cette section, nous allons nous concentrer sur le problème de construire automatiquement un ensemble régulier inévitable (voir définition 3.12, page 57) pour le système $\mathcal{S}$. La différence avec la définition de la section 3.6.3 est qu'ici il faut préciser que l'on s'intéresse à des mots clos. En utilisant la notion d'ensemble inévitable, nous pouvons à présent donner le théorème correspondant dans notre contexte au théorème principal de l'article [BBFM01] (énoncé dans cette thèse en tant que théorème 7.19 à la section 7.4).

Théorème 8.13. *Étant donné un système $\mathcal{S}$ équitable, l'ensemble $Gr((\mathcal{N}_{\mathcal{S}} + \mathcal{R}_{\mathcal{S}})^*)$ est inévitable pour $\mathcal{S}$.*

Démonstration. Comme le système $\mathcal{S}$ est équitable, en particulier $\mathcal{S} - Top_{\mathcal{S}}$ est nœtherien. On peut donc reprendre le raisonnement utilisé pour la preuve du théorème 7.19, réordonner les réductions actives et inactives, afin de ne considérer que des chaînes de surréductions issues de règles de $Top_{\mathcal{S}}$. Celles-ci vont engendrer tous les mots de $(\mathcal{N}_{\mathcal{S}} + \mathcal{R}_{\mathcal{S}})^*$. Comme on suppose le système équitable, le long de toute dérivation infinie toute position est réécrite. En particulier sur un mot du premier ordre, il n'existe pas de dérivation close infinie (c'est-à-dire de dérivation infinie sans surréductions). Toute dérivation infinie contient donc soit une infinité de surréductions, soit une surréduction close.
Comme tout mot clos du premier ordre sert à représenter des mots finis, cela n'a pas de sens d'effectuer une infinité de substitutions de la variable. La seule possibilité est donc d'appliquer à un moment une surréduction close qui fera disparaître la variable. En résumé, toute exécution infinie, après réorganisation des réductions actives et inactives, peut être représentée par une

chaîne de réductions et surréductions issue de top telle que les configurations de la véritable dérivation sont des instances des mots de la chaîne de surréduction. On vient de voir que la chaîne de surréduction passe par $Gr((\mathcal{N}_{\mathcal{S}}+\mathcal{R}_{\mathcal{S}})^*)$, et cet ensemble ne contient que des mots clos (qui sont leur propre instance). On en déduit donc que toute dérivation "réorganisée" infinie passe par l'ensemble $Gr((\mathcal{N}_{\mathcal{S}}+\mathcal{R}_{\mathcal{S}})^*)$. Il faut encore montrer que la véritable dérivation infinie passe également par cet ensemble. Ceci est dû au fait que, comme on effectue des substitutions minimales le long de la chaîne de surréduction, toute la partie close des mots du premier ordre considérés correspond à la partie active des mots de la dérivation originale. Ainsi quand on applique Gr, la partie active couvre tout le mot, et toutes les réductions à partir de ce point sont actives, et ne sont donc pas permutées.

On en déduit donc que le mot de $Gr((\mathcal{N}_{\mathcal{S}}+\mathcal{R}_{\mathcal{S}})^*)$ par lequel passe la chaîne de surréductions issue de *Top* appartient réellement à la dérivation originale, et que $Gr((\mathcal{N}_{\mathcal{S}}+\mathcal{R}_{\mathcal{S}})^*)$ est inévitable. $\square$

Remarque : La différence avec le résultat de [BBFM01] est qu'ici nous ne détectons pas les cycles mais nous calculons un ensemble inévitable qui est composé de mots clos, c'est-à-dire dans lequel la variable a été totalement instanciée et a disparu. Pour s'assurer qu'à toute dérivation infinie va correspondre une suite de surréduction issue de *Top* qui instancie totalement la variable W, il faut supposer que le système de réécriture est équitable.

8.3.1 Système équitable *vs* système nœtherien

Dans l'article de Beauquier *et al.*, l'équité n'était nécessaire que pour l'algorithme d'Hoepman (à trois lettres) où le système est symétrique et où la numérotation des processus est faite de manière arbitraire. Dans notre cadre, si l'on suppose uniquement que $\mathcal{S} - Top$ est nœtherien, on pourrait *a priori* avoir des dérivations infinies qui effectuent un nombre fini de surréductions, sans jamais appliquer de réduction de *Bottom* par exemple, et telles que la variable W ne disparaît pas.

Montrons sur un exemple que l'hypothèse $\mathcal{S} - Top$ nœtherien n'est pas suffisante pour appliquer notre théorème.

Exemple 8.14. Considérons le système suivant. L'alphabet est $\Sigma = \{0,1,2,3,4\}$, et l'ensemble de règles est :

$$\begin{array}{llcll}
\textit{Bottom} & B\ : 13X & \to & 00X & \\
\textit{Top} & T\ : 1X4 & \to & 1X2 & \\
\textit{Middle} & M_1\ : X02Y & \to & X21Y & (\text{ avec } X \neq \varepsilon) \\
 & M_2\ : X32Y & \to & X34Y & (\text{ avec } X \neq \varepsilon) \\
 & M_3\ : X41Y & \to & X04Y & (\text{ avec } X \neq \varepsilon)
\end{array}$$

Ce système vérifie la condition $\mathcal{S} - Top$ est nœtherien[3]. Cependant, il existe des dérivations closes infinies qui ne passent pas par $Gr((\mathcal{N}_{\mathcal{S}}+\mathcal{R}_{\mathcal{S}})^*)$. En effet, une étude un peu plus précise

[3] Pour s'en convaincre, on peut remarquer que le nombre de processus dans l'un des états 2 ou 4 est constant, que les 2 peuvent se transformer en 4 au moyen de la règle M_2 mais pas l'inverse vu qu'on a ôté la règle T. La règle B est appliquée au plus une fois. Ainsi les 2 se déplacent vers la gauche jusqu'à rencontrer un processus dans l'état 3. Ils sont alors changés en 4 et repartent dans l'autre sens jusqu'à être bloqués car ils doivent s'arrêter au plus tard au niveau de la position top. Comme toutes les règles sauf B réécrivent un processus dans l'état 2 ou 4, on ne va pouvoir appliquer qu'un nombre fini de règles.

du système montre que $Gr((\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^*)$ ne contient que des configurations dans lesquelles aucun processus n'est dans l'état 3. Or si l'on considère la dérivation

$$13304 \underset{T}{\rightarrow} 13302 \underset{M_1}{\longrightarrow} 13321 \underset{M_2}{\longrightarrow} 13341 \underset{M_3}{\longrightarrow} 13304$$

elle est quasi-cyclique et peut donc être prolongée en une dérivation infinie, qui ne passe jamais par l'ensemble $Gr((\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^*)$. Si on considère la chaîne de surréductions et de dérivations correspondante,

$$1W4 \underset{T}{\rightarrow} 1W2 \rightsquigarrow_{M_1} 1W21 \rightsquigarrow_{M_2} 1W341 \underset{M_3}{\longrightarrow} 1W304 \underset{T}{\rightarrow} 1W302 \underset{M_1}{\longrightarrow} ... \underset{M_3}{\longrightarrow} 1W304 \underset{T}{\rightarrow} ...$$

on constate que l'on a une dérivation infinie au cours de laquelle on n'applique jamais de surréduction close, et c'est pourquoi on n'arrive jamais dans $Gr((\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^*)$. Ce système ne vérifie donc pas les hypothèses de notre théorème. ♣

Le fait de ne considérer que des systèmes équitables est par définition plus restrictif que le caractère nœtherien de $\mathcal{S} - Top$. En effet, il ne suffit plus de vérifier qu'au moins une règle de *Top* est appliquée infiniment souvent, mais il faut prouver que chaque position est réécrite infiniment souvent.

Ce critère d'équité est souvent vérifié en pratique, en particulier dans les algorithmes de circulation de jeton, comme ceux de Ghosh et Beauquier-Debas.
Il faut remarquer cependant que, pour prouver l'équité, il faut en général trouver à la main une mesure bien fondée spécifique qui décroît à chaque étape ne réécrivant pas la position souhaitée, et ce pour chaque position, pas uniquement pour celle la plus à droite.

Exemple 8.15. On peut montrer que le système $\mathcal{BD}$ de l'exemple 7.2 est équitable. En effet, les règles de $\mathcal{BD}$ préservent le nombre de processus dans l'état 1 et dans l'état 2, et déplacent les 1 dans le sens des positions croissantes, et les 2 dans le sens des positions décroissantes. Si le long d'une dérivation une certaine position n'est jamais réécrite, alors elle va "bloquer le passage" de ces 1 et 2, et au bout d'un moment aucune règle ne sera applicable. Formellement, si on considère la mesure $\psi'_{\mathcal{BD},j} = \sum_{q_i=1}(j-i) \bmod N + \sum_{q_i=2}(i-j) \bmod N$, cette mesure décroît strictement pour toute réduction qui ne réécrit pas le processus à la position j.

♣

8.3.2 Retour sur les ensembles inévitables

La construction d'un ensemble inévitable consiste donc à calculer l'ensemble $(\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^*$ puis lui appliquer une étape de surréduction close. Dans [BBFM01] et [Mag02], les auteurs calculent des sur-approximations de $(\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^*$, sous forme de langages réguliers. Cependant, le processus ne peut pas être totalement automatisé car les sur-approximations doivent être calculées à la main, et spécifiquement pour chaque exemple. Notre travail constitue donc une étape supplémentaire vers l'automatisation de ces preuves.
En effet, nous donnons une condition suffisante simple qui assure que l'ensemble $(\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^*$ est régulier et peut être calculé de manière à la fois exacte et automatique.

Pour la suite de notre méthode, nous allons utiliser le lemme suivant :

Lemme 8.16. *Supposons l'ensemble $\mathcal{N}_\mathcal{S}^*$ fermé pour le système $\mathcal{S}$. Dans ce cas l'ensemble $(\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^*$ est tout simplement $\mathcal{N}_\mathcal{S}^*$.*

Démonstration. La preuve se fait par induction sur le nombre d'alternances entre réductions et surréductions. Soit u un mot appartenant à $(\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^*$. Il existe un entier i tel que $u \in (Reduce^* \circ Narrow^*)^i(I_\mathcal{S})$. Or,

$$\begin{aligned}(Reduce^* \circ Narrow^*)^i(I_\mathcal{S}) &= (Reduce^* \circ Narrow^*)^{i-1}(Reduce^*(\mathcal{N}_\mathcal{S}*)) \\ &= (Reduce^* \circ Narrow^*)^{i-1}(\mathcal{N}_\mathcal{S}*) && \text{(par hypothèse)} \\ &= (Reduce^* \circ Narrow^*)^{i-1}(I_\mathcal{S}) && \text{(par regroupement)}\end{aligned}$$

Ainsi, on peut diminuer le nombre d'alternances, jusqu'à obtenir $u \in \mathcal{N}_\mathcal{S}{}^*$. □

En utilisant le lemme 8.16, le théorème 8.13 devient :

Théorème 8.17. *Soit $\mathcal{S}$ un système de réécriture équitable. Si l'ensemble $\mathcal{N}_\mathcal{S}^*$ est fermé pour $\mathcal{S}$, alors $\mathcal{G}_\mathcal{S}$ est un ensemble régulier inévitable.*

Comme tester la fermeture de $\mathcal{N}_\mathcal{S}^*$ est décidable (car $\mathcal{N}_\mathcal{S}^*$ est régulier), on a ainsi un critère décidable pour trouver un ensemble inévitable.

Exemple 8.18. Pour le système $\mathcal{BD}$ de l'exemple 7.2, on a $\mathcal{N}_{\mathcal{BD}}^* = 1W20^* \cup\ 20^*1W20^*$. Cet ensemble est fermé pour le système $\mathcal{BD}$ qui est équitable, comme on l'a vu à l'exemple précédent. Ainsi, on peut appliquer le théorème ci-dessus et l'ensemble $\mathcal{G}_{\mathcal{BD}} = 210^*$ est inévitable. ♣

Même lorsque $\mathcal{N}_\mathcal{S}^*$ est fermé pour $\mathcal{S}$, l'ensemble $\mathcal{G}_\mathcal{S}$ ne l'est pas en général. Les dérivations infinies passent donc toutes par l'ensemble $\mathcal{G}_\mathcal{S}$ sans pour autant y rester. Si de plus on dispose d'un ensemble $\mathcal{L}$, qui contient $\mathcal{G}_\mathcal{S}$ et qui est fermé pour $\mathcal{S}$, alors toutes les dérivations infinies passent un jour par $\mathcal{L}$ pour ne plus jamais en sortir. On en déduit donc le corollaire suivant :

Corollaire 8.19. *Considérons un système de réécriture équitable $\mathcal{S}$, et un ensemble de configurations $\mathcal{L}$ fermé pour $\mathcal{S}$. Si $\mathcal{N}_\mathcal{S}^*$ est fermé pour $\mathcal{S}$ et si $\mathcal{G}_\mathcal{S}$ est inclus dans $\mathcal{L}$, alors on a $Abs(\mathcal{S}, \mathcal{L})$.*

Démonstration. Soit $\mathcal{S}$ un système de réécriture et $\mathcal{L}$ un ensemble de configurations fermé pour $\mathcal{S}$. S'il existe un ensemble inévitable $\mathcal{K}$ pour $\mathcal{S}$ tel que $\mathcal{K} \subseteq \mathcal{L}$, alors on a $Abs(\mathcal{S}, \mathcal{L})$. En effet, comme $\mathcal{K}(\subseteq \mathcal{L})$ est inévitable, $\mathcal{L}$ l'est aussi. Comme $\mathcal{L}$ est également fermé pour $\mathcal{S}$, il est absorbant (voir définition 3.16, page 58). □

On peut remarquer que, lorsque de plus l'ensemble $\mathcal{L}$ est régulier, l'inclusion de $\mathcal{G}_\mathcal{S}$ dans $\mathcal{L}$ est décidable (vu que $\mathcal{G}_\mathcal{S}$ est régulier si $I_\mathcal{S}$ l'est). Le corollaire 8.19 apporte une amélioration conséquente aux résultats présentés dans [BBFM01] car il permet, lorsque $\mathcal{N}_\mathcal{S}{}^*$ est clos, de prouver automatiquement qu'un ensemble est absorbant, et de là entre autres l'auto-stabilisation (voir chapitre 9).

Exemple 8.20. Pour l'algorithme de Beauquier-Debas, l'ensemble $\mathcal{N}_{\mathcal{BD}}^* = 1W20^* \cup 20^*1W20^*$ est fermé pour $\mathcal{BD}$ et $\mathcal{G}_{\mathcal{BD}} = 210^*$ est inclus dans $\mathcal{L} = 20^*10^* \cup 10^*20^*$. Comme on a vu que de plus le système $\mathcal{BD}$ est équitable et que $\mathcal{L}$ est fermé, on a $Abs(\mathcal{BD}, \mathcal{L})$.
La construction de l'ensemble $\mathcal{G}_{\mathcal{BD}}$ peut être faite automatiquement par un programme que nous présenterons à la section 9.3. ♣

Ainsi il n'est plus nécessaire de construire ni de manipuler des graphes de dérivations. La condition permettant d'appliquer le corollaire 8.19 limite les cas d'application, mais lorsque ces conditions sont réunies, on est sûr de parvenir à démontrer que l'ensemble $\mathcal{L}$ considéré est absorbant.

8.4 Condition suffisante de fermeture de ${\mathcal{N}_\mathcal{S}}^*$

Même si la fermeture de ${\mathcal{N}_\mathcal{S}}^*$ du corollaire 8.19 est décidable, on ne peut pas vérifier par simple inspection des règles si elle est satisfaite ou non. C'est pourquoi nous avons cherché une condition suffisante pour assurer cette fermeture, et qui soit vérifiable simplement, dès lors qu'on connaît les règles.

Afin d'établir notre critère, nous avons besoin de quelques définitions préliminaires :

Définition 8.21. *Un* demi-tour à gauche *(resp. un* demi-tour à droite*) dans une dérivation close est une suite de réductions de la forme* $u\mathbf{abc}v \to ua'\mathbf{b'c}v \to u\mathbf{a'b''}c'v \to ua''b'''c'v$ *(resp.* $ua\mathbf{bc}v \to u\mathbf{ab'}c'v \to ua'\mathbf{b''c'}v \to ua'b'''c''v$*), avec* u *et* $v \in \Sigma^*$

Définition 8.22. *Un* demi-tour bottom *dans une dérivation close est une suite de réductions de la forme* $\mathbf{abc}v \to a'\mathbf{b'c}v \to \mathbf{a'b''}c'v \to a''b'''c'v$. *Un* demi-tour top à droite *(resp.*demi-tour top à gauche*) est une suite de réductions de la forme* $\mathbf{c}ua\mathbf{b} \to c'u\mathbf{ab'} \to c'ua'b''$ *tel qu'il existe* $d \in \Sigma, v \in \Sigma^+$ *tels que* $\mathbf{d}v\mathbf{b''} \to d'vb'''$ *(resp.* $\mathbf{c}au\mathbf{b} \to \mathbf{c'a}ub' \to c''a'ub'$ *tel qu'il existe* $d \in \Sigma, v \in \Sigma^+$ *tels que* $\mathbf{c''}v\mathbf{d} \to c'''vd'$ *).*

Ces définition prennent en compte le cas de la topologie en anneau. Lorsqu'on a une topologie linéaire, il faut remplacer les demi-tour top gauche et droite par un seul type de demi-tour top :

$$va\mathbf{bc} \to v\mathbf{ab'}c' \to va'\mathbf{b''c'} \to va'b'''c''$$

Ces définitions s'adaptent naturellement pour des règles à trois lettres. La particularité des demi-tours top à droite (resp. à gauche) est qu'on laisse la possibilité d'autres réductions entre la deuxième et la troisième réduction du demi-tour. En effet, la lettre en première (resp. dernière) position peut avoir changé.

Enfin, la dernière suite particulière de réductions que nous allons considérer est plus simple, c'est la répétition.

Définition 8.23. *Une* répétition *dans une dérivation close est une suite de deux réductions consécutives qui s'appliquent à la même position, par exemple* $u\mathbf{ab}v \to u\mathbf{a'b'}v \to ua''b''v$

Un système de réécriture $\mathcal{S}$ est dit *unidirectionnel* si aucun demi-tour ou répétition n'est possible le long des dérivations de $\mathcal{S}$.
L'intérêt de ce critère est qu'il est facile de décider, rien qu'en regardant les membres droit et gauche des règles, si le système $\mathcal{S}$ est ou non unidirectionnel.

Proposition 8.24. *Pour un système de réécriture* $\mathcal{S}$ *unidirectionnel, on a* $(\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^* = \mathcal{N}_\mathcal{S}^*$.

Démonstration. La preuve se fait par l'absurde. On va montrer qu'il n'est pas possible d'appliquer une réduction close à un mot de $\mathcal{N}_\mathcal{S}^*$ (c'est-à-dire engendré par surréductions à partir de $I_\mathcal{S}$ sans application de surréduction close). Supposons qu'il existe des mots du premier ordre u, v et w, et une règle R de $\mathcal{S}$ tels que $I_\mathcal{S} \ni w \rightsquigarrow_\mathcal{S}^* v \underset{R}{\to} u$. Ici le mot v est donc engendré à partir de u au moyen uniquement de surréductions (utilisant des instanciations non triviales), donc $v \in {\mathcal{N}_\mathcal{S}}^*$. Dans cette dérivation, la règle R est appliquée sur la partie close de $v = v_1 W v_2$. Il y a deux cas principaux :

- si la règle est appliquée à des lettres les plus proches de la variable – par exemple à ab où $v_1 = v_1'ab$ – alors c'est impossible car cela correspondrait à une répétition. En effet, si $v_1 = v_1'ab$, le motif ab a été obtenu en appliquant une surréduction du type $v_1'a'Wv_2' \rightsquigarrow_{R'} v_1'abWv_2'$. Les règles R' et R peuvent donc être appliquées successivement à la même position, ce qui correspond à une répétition.

– si la règle est appliquée ailleurs – par exemple au facteur ab tel que $v_2 = v_2' ab v_2''$ – alors on peut permuter des applications de règles pour montrer que cela correspond à un demi tour (à droite, à gauche, top ou bottom selon le cas). Montrons-le sur l'exemple ci-dessus. Comme les surréductions à droite d'une part, à gauche et bottom d'autre part sont indépendantes, on ne va considérer que des surréductions à droite. Ainsi, la dérivation qui a mené à u contient des surréductions du genre $t_1 W b' t_2 \rightsquigarrow_{R_1} t_1 W a' b t_2 \rightsquigarrow_{R_2} t_1 W cab t_2$, et plus tard $t_1 W t_3 ab t_2 \underset{R}{\rightarrow} t_1 W t_3 a'' b'' t_2$. Cela implique qu'il existe un demi-tour à droite

$$u_1 c' \mathbf{a}''' \mathbf{b}' u_2 \underset{R_1}{\longrightarrow} u_1 \mathbf{c}' \mathbf{a}' b u_2 \underset{R_2}{\longrightarrow} u_1 c \mathbf{ab} u_2 \underset{R}{\rightarrow} u_1 c a'' b'' u_2$$

ce qui est impossible par hypothèse.

Ainsi, le fait que le système soit unidirectionnel empêche l'application de surréductions closes aux mots de $\mathcal{N}_{\mathcal{S}}^*$. On en déduit donc que $(\mathcal{N}_{\mathcal{S}} + \mathcal{R}_{\mathcal{S}})^* = \mathcal{N}_{\mathcal{S}}^*$. □

Le fait d'être unidirectionnel est donc une condition aisément vérifiable qui garantit que $\mathcal{N}_{\mathcal{S}}^*$ est fermé pour $\mathcal{S}$.

Tous les systèmes ne sont bien sûr pas unidirectionnels. Cependant, nous pensons qu'un système $\mathcal{S}$ peut souvent être transformé en un système simplifié $\mathcal{T}$ qui est unidirectionnel, et tel que $Abs(\mathcal{T}, \mathcal{L})$ implique $Abs(\mathcal{S}, \mathcal{L})$. Cette simplification se fait en général en considérant une variante de $\mathcal{S}$ qui préserve une certaine fonction non croissante φ, comme décrite à la section 7.5.

Si on s'intéresse au cas des algorithmes d'exclusion mutuelle, où l'on converge vers un ensemble de configurations qui ne contiennent plus qu'un seul jeton, la fonction φ compte en général le nombre de jetons d'une configuration.

Exemple 8.25. En ce qui concerne l'algorithme de Beauquier-Debas, la version simplifiée $\mathcal{BD}$ (décrite à l'exemple 7.2, page 121) du système vérifie la condition d'unidirectionnalité, alors que la version originale présentée à la section 7.8.1 n'est pas unidirectionnelle. Par exemple, elle permet le demi-tour suivant $u1\mathbf{10}v \rightarrow_{M_1} u\mathbf{10}1v \rightarrow_{M_1} u0\mathbf{11}v \rightarrow_{M_2} u002v$. ♣

Remarque : Bien évidemment, la condition exhibée dans cette section est suffisante, mais pas nécessaire pour assurer la fermeture de $\mathcal{N}_{\mathcal{S}}^*$.

Cependant, cette condition nous permet de mieux comprendre pourquoi notre hypothèse de fermeture de $\mathcal{N}_{\mathcal{S}}^*$ est vérifiée en pratique par des algorithmes d'exclusion mutuelle auto-stabilisante, comme par exemple ceux de Beauquier-Debas et Ghosh [BD95, Gho93]

Chapitre 9

Applications et résultats

Nous allons maintenant présenter deux applications du corollaire 8.19. Tout au long de cette section, nous allons faire trois hypothèses à propos de $\mathcal{S}$ et $\mathcal{L}$:

α. L'ensemble $\mathcal{L}$ est régulier ;
β. L'ensemble $\mathcal{L}$ est fermé pour $\mathcal{S}$;
γ. Le système $\mathcal{S}$ est équitable.

Nous montrons tout d'abord comment prouver l'auto-stabilisation, puis une propriété de vivacité pour les algorithmes de détection de terminaison.

9.1 Preuve d'auto-stabilisation

Dans le contexte de ce chapitre, prouver l'auto-stabilisation consiste à montrer deux choses :

1. Toute configuration close n'appartenant pas à $\mathcal{L}$ est réductible pour $\mathcal{S}$;
2. l'ensemble $\mathcal{L}$ est absorbant pour $\mathcal{S}$.

La première condition est décidable. En effet, elle consiste à inverser les membres droit et gauche des règles, et appliquer une étape de réduction du système inversé. Tout mot hors de $\mathcal{L}$ est réductible si et seulement si tout mot hors de $\mathcal{L}$ est inclus dans $Red^{-1}(\Sigma^*)$. Le problème principal est donc de prouver la deuxième propriété, représentée à la figure 9.1.

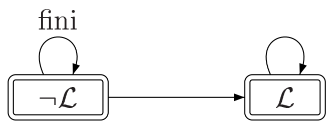

Fig. 9.1 – Graphe d'exécution pour les systèmes auto-stabilisants

Pour les algorithmes auto-stabilisants sur les anneaux, l'ensemble légitime peut souvent être exprimé de manière naturelle comme un ensemble régulier. Cela implique que la condition α est souvent satisfaite, et la condition β est décidable. La condition γ est également assez souvent vérifiée. Reste à s'assurer que l'ensemble $\mathcal{N}_{\mathcal{S}}^*$ est fermé pour $\mathcal{S}$, et pour cela il faut souvent considérer un système simplifié correspondant à $\mathcal{S}$, comme on l'a vu à la section 8.4.

9.1.1 L'algorithme de Beauquier-Debas

Pour la dernière fois, nous allons revenir sur l'algorithme d'exclusion mutuelle dû à Beauquier-Debas ([BD95]), adaptation de l'algorithme de Dijkstra à 3 états ([Dij74]).

Rappelons tout d'abord le problème. Le système $\mathcal{S}$ représentant l'algorithme original de Beauquier-Debas est le suivant :

$$\begin{array}{ll}
B_1 :\ 12X \to 21X & T_1 : 0X0 \to 1X2 \\
M_1 : X10Y \to X01Y\ \ (X \neq \varepsilon) & T_2 : 0X1 \to 1X0 \\
M_2 : X11Y \to X02Y\ \ (X \neq \varepsilon) & T_3 : 0X2 \to 1X1 \\
M_3 : X12Y \to X00Y\ \ (X \neq \varepsilon) & T_4 : 2X1 \to 1X2 \\
M_4 : X02Y \to X20Y\ \ (X \neq \varepsilon) & T_5 : 2X2 \to 1X0 \\
M_5 : X22Y \to X10Y\ \ (X \neq \varepsilon) &
\end{array}$$

L'ensemble $\mathcal{L}$ est défini par l'expression régulière $20^*10^* \cup 10^*20^*$. Il est possible (en calculant automatiquement $\mathcal{N}_{\mathcal{S}}^*$ et en testant une inclusion) de montrer que $\mathcal{N}_{\mathcal{S}}^*$ n'est pas fermé pour $\mathcal{S}$, c'est pourquoi nous allons appliquer notre méthode à un système simplifié, construit à partir de $\mathcal{S}$ comme annoncé à la section 7.8.1.
Tout d'abord on va laisser de côté les règles T_1, T_2 et T_3 qui ne peuvent être appliquées qu'au plus une fois, puis les règles M_2, M_3, M_5 et T_5 qui ne préservent pas la mesure $\varphi_{\mathcal{BD}}$ décrite à la section 7.8.1. Il ne reste donc plus que quatre règles, et comme les autres ne peuvent être appliquées qu'un nombre fini de fois le long d'une dérivation, on en déduit que l'ensemble $\mathcal{L}$ est absorbant pour $\mathcal{S}$ si et seulement si il est absorbant pour $\mathcal{BD} \equiv \{B_1, M_1, M_4, T_4\}$.
Comme cela a été montré au cours des exemples de ce chapitre, le système $\mathcal{BD}$ est unidirectionnel, l'ensemble $Gr(\mathcal{N}_{\mathcal{BD}}{}^*)$ vaut 210^* et est inclus dans $\mathcal{L}$ qui est clos. On a donc $Abs(\mathcal{BD}, \mathcal{L})$, et on en déduit que $Abs(\mathcal{S}, \mathcal{L})$. Comme de plus il n'y a pas de blocage hors de $\mathcal{L}$, le système original $\mathcal{S}$ est auto-stabilisant.

9.1.2 L'algorithme de Ghosh

L'algorithme d'exclusion mutuelle de Ghosh [Gho93], présenté à la section 7.8.2, se compose à l'origine de règles à deux lettres. Cependant, l'utilisation de la mesure $\varphi_{\mathcal{G}}$ permet de se ramener à un système conservant cette mesure. Cela change relativement le système étant donné que les règles de $Middle_{\mathcal{G}}$ impliquent trois lettres consécutives. L'alphabet est $\Sigma = \{0, 1, 2, 3\}$ et les règles du système simplifié $\mathcal{G}$ sont :

$$\begin{array}{rl}
C_1 : & 12X \to 32X \\
C_2 : & 30X \to 10X \\
D_1(q) : & Xqq(q+1)Y \to Xq(q+1)(q+1)Y \\
D_2(q) : & X(q+1)qqY \to X(q+1)(q+1)qY \\
E_1 : & X32 \to X30 \\
E_2 : & X10 \to X12
\end{array}$$

L'ensemble $\mathcal{L}$ des configurations légitimes est $3^+0^+ \cup 3^+2^+ \cup 1^+2^+ \cup 1^+0^+$. On a vu à la section 7.8.2 que $\mathcal{G} - Top_{\mathcal{G}}$ est nœtherien. Le même raisonnement peut, sans grande modification, s'appliquer à toute position pour montrer que dans une dérivation close, si une position n'est jamais réécrite, alors la dérivation est finie. Le système $\mathcal{G}$ est donc bien équitable.

Par construction du système $\mathcal{G}$ (= variante du système original $\mathcal{S}$ préservant la mesure $\varphi_{\mathcal{G}}$ non croissante), on a $Abs(\mathcal{G}, \mathcal{L}) \Rightarrow Abs(\mathcal{S}, \mathcal{L})$. Comme pour l'algorithme de Beauquier-Debas, l'algorithme de Ghosh vérifie les conditions α, β et γ, ainsi que l'unidirectionnalité. On peut ainsi appliquer notre méthode (adaptée au cas des règles à trois lettres).

Le calcul du langage $Gr(\mathcal{N}_{\mathcal{G}}{}^*)$[1] donne $100^+ \cup 322^+$. Comme cet ensemble est inclus dans $\mathcal{L}$, on en déduit que $\mathcal{L}$ est absorbant pour $\mathcal{G}$ donc pour $\mathcal{S}$ d'où l'auto-stabilisation.

9.2 Preuve de vivacité pour la détection de terminaison

Comme on l'a vu à la section 3.1.4, un système distribué termine lorsqu'il n'y a plus d'actions possibles dans une configuration, ou du moins plus aucune action "intéressante". Cependant, même si la configurations globale est terminale, certains processus peuvent ne pas en avoir connaissance. Un algorithme de détection de terminaison est un algorithme qui observe l'exécution du système original et détecte si celui-ci a atteint une configuration terminale.

Parmi les propriétés que doit vérifier un algorithme de détection de terminaison, il y a la vivacité, à savoir : "si à un moment la configuration est terminale, alors l'algorithme d'annonce doit être appelé en un temps fini." L'algorithme d'annonce servant dans ce cas à prévenir tous les processus que la configuration est terminale, et ainsi les mettre dans un état final.

Dans notre contexte, étant donné un ensemble *Term* (fermé pour $\mathcal{S}$) contenant toutes les configurations terminales, la propriété de vivacité $Liv(\mathcal{S}, Term)$ peut être reformulée comme suit : lorsque l'ensemble *Term* est atteint, alors l'algorithme termine (ou appelle la fonction d'annonce) en un temps fini. Cette propriété est représentée à la figure 9.2.

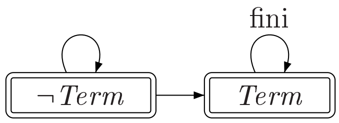

FIG. 9.2 – Graphe d'exécution du système $\mathcal{S}$ satisfaisant $Liv(\mathcal{S}, Term)$

Formellement :

Définition 9.1. *Soit $\mathcal{S}$ un système de réécriture, et soit Term un ensemble de configurations fermé pour $\mathcal{S}$. On dit que le système $\mathcal{S}$ satisfait la propriété $Liv(\mathcal{S}, Term)$ si toute dérivation close partant d'un élément de Term est finie.*

La méthode décrite précédemment ne s'applique pas directement à la preuve de la propriété $Liv(\mathcal{S}, Term)$, mais peut être adaptée comme suit.

On considère tout d'abord le système inversé $\mathcal{S}^{-1}$, c'est-à-dire qu'on échange, pour chaque règle de $\mathcal{S}$, le membre droit et le membre gauche. Si l'on considère à présent les dérivations suivant le système $\mathcal{S}^{-1}$, la propriété de vivacité correspond au graphe d'exécution représenté à la figure 9.3.

En effet, prouver que toute exécution dans *Term via* $\mathcal{S}$ est finie revient à prouver que toute exécution *via* $\mathcal{S}^{-1}$ va sortir de *Term*, c'est-à-dire $Abs(\mathcal{S}^{-1}, \neg Term)$.

Si l'on suppose que le système $\mathcal{S}$ et l'ensemble *Term* satisfont α, β et γ, alors les conditions suivantes sont satisfaites :

α'. l'ensemble $\neg Term$ est régulier (car *Term* est régulier),

[1]Dans ce cas il n'est pas utile d'appliquer préfixe et suffixe en parallèle vu que les extrémités du réseau ne communiquent pas.

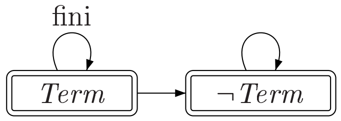

FIG. 9.3 – Graphe d'exécution du système inversé $\mathcal{S}^{-1}$ satisfaisant $Abs(\mathcal{S}^{-1}, \neg\, Term)$

β'. l'ensemble $\neg\, Term$ est fermé pour $\mathcal{S}^{-1}$ (car $Term$ est fermé pour $\mathcal{S}$),
γ'. le système $\mathcal{S}^{-1}$ est équitable (car $\mathcal{S}$ l'est et car les règles de $\mathcal{S}$ préservent la longueur).

Nous pouvons donc utiliser notre méthode pour prouver $Abs(\mathcal{S}^{-1}, \neg\, Term)$, et en déduire $Liv(\mathcal{S}, Term)$. Formellement, on a le théorème suivant :

Théorème 9.2. *Étant donnés un système équitable $\mathcal{S}$ et un ensemble de configurations Term, régulier et fermé pour $\mathcal{S}$. Si l'ensemble $\mathcal{N}^*_{\mathcal{S}^{-1}}$ est fermé pour $\mathcal{S}^{-1}$ et si $\mathcal{G}_{\mathcal{S}^{-1}}$ est inclus dans l'ensemble $\neg\, Term$, alors on a la propriété $Liv(\mathcal{S}, Term)$.*

Tester la fermeture de $\mathcal{N}^*_{\mathcal{S}^{-1}}$ et l'inclusion de $\mathcal{G}_{\mathcal{S}^{-1}}$ est décidable, étant donné que ces deux ensembles sont réguliers.

La détection de terminaison concerne souvent un ensemble plus restreint de configurations dites "admissibles", comme par exemple les configurations avec au plus un jeton. En notant Adm l'ensemble (régulier) de tous les mots clos admissibles, le théorème 9.2 est dans ce cas légèrement modifié, en remplaçant $N^*_{\mathcal{S}^{-1}}$ par $N^*_{\mathcal{S}^{-1}} \cap Adm$.

9.2.1 L'algorithme de Dijkstra-Feijen-van Gasteren

Nous avons appliqué cette méthode pour prouver la vivacité de l'algorithme de détection de terminaison de Dijkstra, Feijen et van Gasteren [DFvG83] (voir aussi [Tel94, JN00])

Dans cet algorithme, l'état d'un processus est représenté par un couple. La première composante décrit la situation dans laquelle se trouve ce processus : ac s'il est actif et $inac$ sinon. La deuxième composante indique si le processus possède un jeton ou non : elle vaut e lorsque le processus n'a pas de jeton, b (black) s'il a un jeton de couleur noire, et w (white) s'il a un jeton de couleur blanche. Nous allons parfois utiliser la lettre t, qui est signifie que le processus possède un jeton, sans préciser sa couleur. Ainsi (ac, t) désigne en fait deux états : (ac, b) et (ac, w).

Le principe de l'algorithme est le suivant : un processus actif peut à tout moment devenir inactif. Par contre, pour qu'un processus devienne actif, il faut que son voisin de gauche soit également actif. En ce qui concerne le jeton, il peut passer d'un processus à l'autre de gauche à droite (= dans le sens des indices croissants). Lorsqu'il croise un processus inactif, il conserve sa couleur, mais dès qu'il a croisé un processus actif, il devient noir.

Considérons l'ensemble de règles simplifié suivant, que nous appellerons $\mathcal{DFG}$:

$$\begin{array}{llll}
\textit{Bottom} & B_1: & (ac,t)(_,e)X \rightarrow (_,e)(_,b)X & \\
 & B_2: & (inac,b)(inac,e)X \rightarrow (inac,e)(inac,w)X & \\
 & B_3: & (inac,b)(ac,e)X \rightarrow (inac,e)(_,w)X & \\
\textit{Middle} & M_1: & X(ac,t)(_,e)Y \rightarrow X(_,e)(_,b)Y & (X \neq \varepsilon) \\
 & M_2: & X(inac,t)(inac,e)Y \rightarrow X(inac,e)(inac,t)Y & (X \neq \varepsilon) \\
 & M_3: & X(inac,t)(ac,e)Y \rightarrow X(inac,e)(_,t)Y & (X \neq \varepsilon) \\
\textit{Top} & T_1: & (_,e)X(ac,t) \rightarrow (_,b)X(_,e) & \\
 & T_2: & (inac,e)X(inac,t) \rightarrow (inac,t)X(inac,e) & \\
 & T_3: & (ac,e)X(inac,t) \rightarrow (_,t)X(inac,e) &
\end{array}$$

Les notations _ et t ci-dessus désignent respectivement un caractère dans $\{ac, inac\}$ et $\{b, w\}$.

Dans cet exemple, l'ensemble *Term* est l'ensemble de toutes les configurations pour lesquelles tous les processus sont inactifs, à savoir $Term = (inac, \{e, b, w\})^+$. Le but est donc de prouver que, une fois qu'on a atteint une configuration de *Term*, alors la fonction d'annonce va être appelée au bout d'un nombre fini d'étapes, c'est-à-dire $Liv(\mathcal{S}, Term)$.

Ce système est une version simplifiée d'une variante (inspirée de [JN00]) de l'algorithme de Dijkstra-Feijen-van Gasteren. En effet, nous ne considérons que des règles qui réécrivent un jeton. Cette simplification est justifiée car, lorsqu'une configuration est dans *Term*, il n'y a pas de processus actif donc typiquement pas de règle du genre $(ac, e) \rightarrow (inac, e)$.

Cette simplification conserve donc la propriété souhaitée, à savoir que $Liv(\mathcal{DFG}, Term)$ entraîne $Liv(\mathcal{S}, Term)$ et $\mathcal{DFG}$ satisfait les propriétés α, β et γ.
Comme annoncé précédemment, nous allons nous intéresser à un ensemble fermé de configurations dites admissibles, à savoir celles qui contiennent au plus un jeton, c'est-à-dire $Adm = (\_,e)^*W(\_,e)^* \cup (\_,e)^*(\_,t)(\_,e)^*W(\_,e)^* \cup (\_,e)^*W(\_,e)^*(\_,t)(\_,e)^*$.

Afin de prouver la propriété $Liv(\mathcal{DFG}, Term)$, nous utilisons notre méthode en considérant le système inversé $\mathcal{DFG}^{-1}$ et en prouvant $Abs(\mathcal{DFG}^{-1}, \neg Term)$. Pour cela nous montrons qu'il existe un ensemble inévitable pour $\mathcal{DFG}^{-1}$ inclus dans $\neg Term$. La preuve est en deux étapes.

Les membres droits des règles *Top* de $\mathcal{DFG}^{-1}$ (donc les membres gauches des règles *Top* de $\mathcal{DFG}$) sont de la forme $u_b = (\_,e)W(\_,b)$ ou $u_w = (\_,e)W(\_,w)$. Comme le système $\mathcal{DFG}^{-1}$ est unidirectionnel, le langage $\mathcal{N}^*_{\mathcal{DFG}^{-1}}$ engendré par surréductions successives est clos. En prenant l'intersection avec *Adm* et en appliquant une étape de surréduction close, on obtient les deux langages $\mathcal{G}_b$ issu de u_b et $\mathcal{G}_w$ issu de u_w, avec :

$$\begin{array}{lcl}
\mathcal{G}_b & = & (ac,t)(_,e)^+ \cup (inac,b)(inac,e)^*(ac,e)(_,e)^+ \\
\mathcal{G}_w & = & (inac,b)(_,e)^+.
\end{array}$$

On peut remarquer que le langage réellement engendré par surréductions successives est plus grand que cela, mais on ne considère que son intersection avec *Adm*.
On peut constater que $\mathcal{G}_b \subset \neg Term$. Ceci n'est pas vrai pour l'ensemble $\mathcal{G}_w$, mais si à partir de $\mathcal{G}_w$ on applique une réduction *via* $\mathcal{DFG}^{-1}$ (au moyen d'une des règles de *Top* : $\{T_1^{-1}, T_2^{-1}, T_3^{-1}\}$, obtient le langage $\mathcal{H}_w \cup \mathcal{H}'_w$, avec $\mathcal{H}_w = (\_,e)^+(ac,t) \subseteq \neg Term$ et $\mathcal{H}'_w = (\_,e)^+(inac,b)$.
Il ne reste donc plus qu'à montrer qu'à partir de $\mathcal{H}'_w$ on va finir par arriver dans $\neg Term$. Or il se trouve que l'ensemble $\mathcal{H}'_w$ se compose d'instances du mot du premier ordre $(\_,e)X(\_,b)$. En faisant une généralisation, on se ramène à $u_b = (\_,e)W(\_,b)$, et en ré-appliquant une fois le processus on montre que toute dérivation est forcée d'arriver dans l'ensemble $\neg Term$.
Toute cette méthode est illustrée sur le schéma de la figure 9.4.

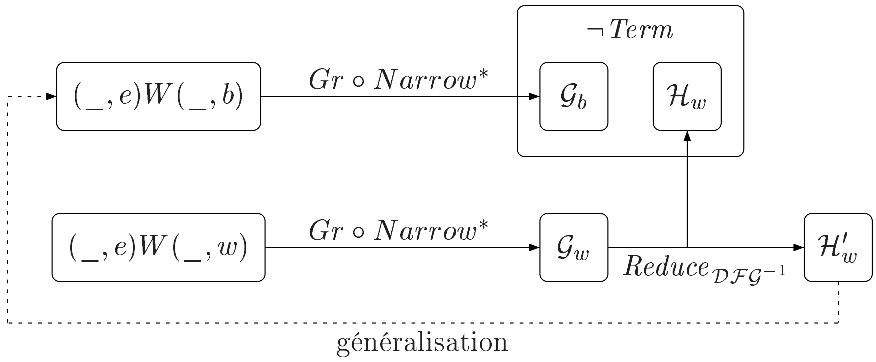

FIG. 9.4 – Schéma illustrant notre preuve de vivacité de l'algorithme $\mathcal{DFG}$

Nous avons donc montré qu'en au plus deux applications de chaînes de surréductions, toute dérivation arrive dans un langage inévitable $\mathcal{K} = \mathcal{G}_b \cup \mathcal{H}_w$. On en déduit $Abs(\mathcal{DFG}^{-1}, \neg\, Term)$, et par conséquent $Liv(\mathcal{DFG}, Term)$.

9.3 Implantation et résultats

9.3.1 Le programme

Notre méthode de preuve de convergence a été implantée au moyen du langage SICStus Prolog, et en utilisant les bibliothèques FSA (Finite State Automata) de van Noord [vN00]. Le programme se compose d'approximativement 400 lignes (100 clauses) de code Prolog, sans compter les bibliothèques.

Le programme utilise l'algorithme de Caucal pour calculer les configurations accessibles par réécriture préfixe et suffixe en partant d'un mot t_0 de la forme $u_0 \odot v_0$ et en utilisant les systèmes $\mathcal{S}_{pre}$ (l'extension préfixe de $\mathcal{S}$), et $\mathcal{S}_{suf}$ (l'extension suffixe de $\mathcal{S}$).

Le programme construit donc des automates finis $A_1(\mathcal{S}_{pre}, t_0)$ (resp. $A_2(\mathcal{S}_{suf}, t_0)$), dont le langage est la fermeture réflexive et transitive de $\rightarrow_{\mathcal{S}_{pre}}$ (resp. $\rightarrow_{\mathcal{S}_{suf}}$) appliquée à $\odot v_0$ (resp. $u_0 \odot$). Il applique ensuite une étape de surréduction close pour obtenir $Gr(\mathcal{N_S}^*)$.

Nous avons utilisé notre programme sur les différents algorithmes présentés dans ce chapitre, à savoir ceux de Beauquier-Debas [BD95], Ghosh [Gho93] et Dijkstra-Feijen-van Gasteren [DFvG83]. L'algorithme de Ghosh utilise des règles à trois lettres, et nous avons donc ajouté cette possibilité dans le programme. L'exécution du programme sur chacun de ces exemples a duré au plus quelques secondes sur un Pentium Pro 200MHz avec 128MB de mémoire. Ce temps d'exécution inclut les opérations de simplification pour faciliter la lecture des automates rendus en sortie.

9.3.2 Les résultats

Le programme rend en sortie une liste d'automates. Chacun d'entre eux est décrit sous la forme `fa(_,N,I,F,T,_)` où `N` est le nombre d'états, `I` l'ensemble des états initiaux, `F` l'ensemble des états finals et `T` l'ensemble des transitions écrites sous la forme :

```
trans(départ,étiquette,arrivée).
```

Résultat pour le système $\mathcal{BD}$

La sortie du programme consiste en un seul automate :

```
X = fa(r(fsa_preds),3,[0],[1],[trans(0,2,2),trans(1,0,1),
trans(2,1,1)],[])
```

Cet automate est représenté à la figure 9.5. Le langage régulier associé est bien 210^*, comme annoncé à la partie 8.2.3.

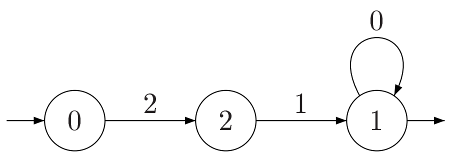

FIG. 9.5 – Automate engendrant le langage $Gr(\mathcal{N}_{\mathcal{BD}}{}^*)$

Résultats pour le système $\mathcal{G}$

Dans le cas de l'algorithme de Ghosh [Gho93], comme les deux extrémités ne communiquent pas, il suffit de calculer le langage engendré par réécriture préfixe à partir d'une règle de $Top_{\mathcal{G}}$. Une variante à trois lettres de notre programme donne en sortie les deux automates suivants :

```
X = fa(r(fsa_preds),4,[0],[1],[trans(0,1,3),trans(1,0,1),trans(2,0,1),
trans(3,0,2)],[])

X = fa(r(fsa_preds),4,[0],[1],[trans(0,3,3),trans(1,2,1),trans(2,2,1),
trans(3,2,2)],[])
```

Ces automates sont représentés à la figure 9.6. Le langage obtenu est donc $100^+ \cup 322^+$, comme annoncé à la section 9.1.2.

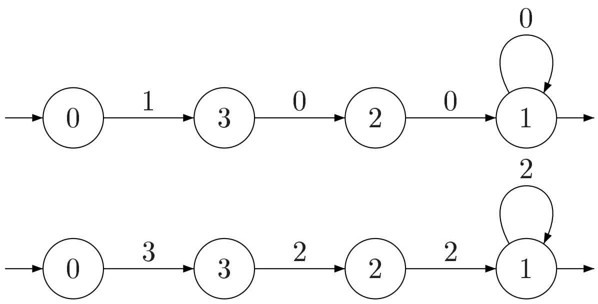

FIG. 9.6 – Automates engendrant le langage $Gr(\mathcal{N}_{\mathcal{G}}{}^*)$

Résultats pour le système $\mathcal{DFG}^{-1}$

Comme toute extension suffixe des règles de $\mathcal{DFG}^{-1}$ va être de la forme

$$X(\_,e)\odot Y \to X(\_,t)(\_,e)\odot Y$$

(éventuellement sans la variable X pour les règles de *Bottom*), on ne peut pas appliquer de telles extensions sans créer de nouveaux jetons. Reprenons les notations de la section 8.2.2 où $\mathcal{M}_0$ (resp. $\mathcal{M}_1$) est le sous-ensemble de $\mathcal{N_S}^*$ obtenu en appliquant aucune (resp. une) surréduction à l'aide d'une règle de $Bottom_{\mathcal{S}}$. Rappelons-nous également que dans le cadre de l'algorithme $\mathcal{DFG}$, on ne va s'intéresser qu'aux configurations de *Adm*, c'est à dire celles n'ayant qu'un seul jeton. On a alors $(\mathcal{M}_0 \cup \mathcal{M}_1) \cap Adm = \mathcal{M}_0$.
Pour calculer $Gr(\mathcal{N}^*_{\mathcal{DFG}^{-1}} \cap Adm)$ il suffit de ne considérer que des surréductions à droite de la variable, c'est à dire correspondant à de la réécriture préfixe.

En prenant en entrée le mot du premier ordre $\#(\_,e)W(\_,w)\#$ et en n'effectuant que des réductions préfixes, la sortie correspondant à $\mathcal{G}_w$ se compose des cinq automates suivants :

```
X = fa(r(fsa_preds),3,[0],[1],[trans(0,'inac+b',2),trans(1,'act+eps',1),
trans(1,'inac+eps',1),trans(2,'act+eps',1)],[])

X = fa(r(fsa_preds),3,[0],[1],[trans(0,'inac+b',2),trans(1,'act+eps',1),
trans(1,'inac+eps',1),trans(2,'inac+eps',1)],[])

X = fa(r(fsa_preds),3,[0],[1],[trans(0,'inac+b',2),trans(2,'act+eps',1)],
[])

X = fa(r(fsa_preds),4,[0],[1],[trans(0,'inac+b',3),trans(1,'act+eps',1),
trans(1,'inac+eps',2),trans(2,'act+eps',1),trans(2,'inac+eps',2),
trans(3,'act+eps',2)],[])

X = fa(r(fsa_preds),4,[0],[1],[trans(0,'inac+b',3),trans(1,'act+eps',1),
trans(1,'inac+eps',2),trans(2,'act+eps',1),trans(2,'inac+eps',2),
trans(3,'inac+eps',2)],[])
```

Ces automates sont représentés à la figure 9.7.

Les langages sont respectivement :

$$L_1 = (inac,b)(ac,e)(\_,e)^*$$
$$L_2 = (inac,b)(inac,e)(\_,e)^*$$
$$L_3 = (inac,b)(ac,e)$$
$$L_4 = (inac,b)(ac,e)(inac,e)^*(ac,e)^+((inac,e)^+(ac,e)^+)^* \text{ et}$$
$$L_5 = (inac,b)(inac,e)(inac,e)^*(ac,e)^+((inac,e)^+(ac,e)^+)^*.$$

On peut remarquer que $L_3 \subset L_1$, $L_4 \subset L_1$ et $L_5 \subset L_2$. Il s'ensuit que le langage $\mathcal{G}_w$ vaut $L_1 \cup L_2$, c'est-à-dire $(inac,b)(\_,e)^+$.

Le langage $\mathcal{G}_b$ est calculé de manière similaire, et on obtient 15 automates.

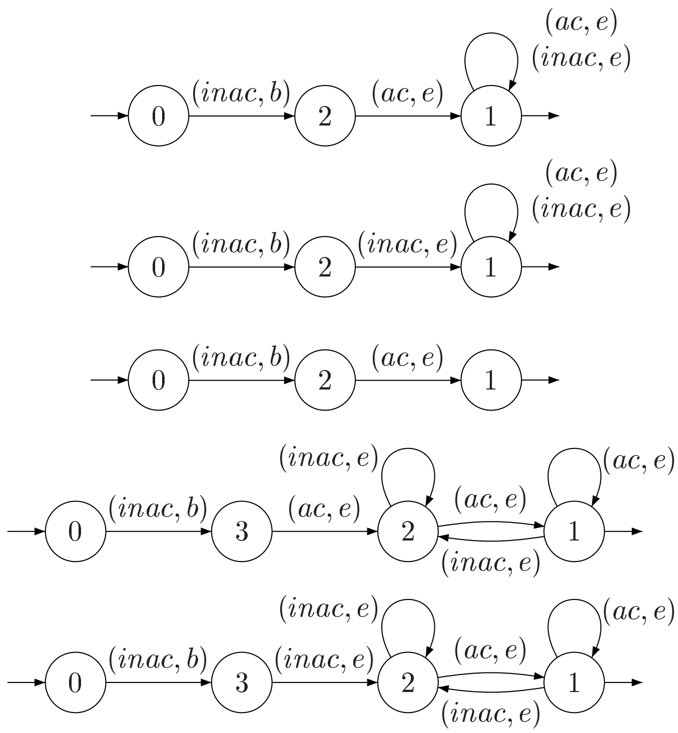

FIG. 9.7 – Automates engendrant le langage $\mathcal{G}_w$

Chapitre 10

Conclusion

Nous avons présenté dans cette thèse deux approches de preuves de convergence sur des anneaux paramétrés.

Systèmes paramétrés probabilistes

La première approche concerne les systèmes paramétrés probabilistes. Elle a consisté à exploiter un résultat de théorie de Markov permettant d'assurer la convergence d'une chaîne de Markov vers un ensemble d'états récurrents. Nous avons adapté ce résultat au cas des processus de décision markoviens, c'est à dire lorsque le système contient à la fois des choix probabilistes et des choix non-déterministes.
Nous avons ensuite montré que, si on exhibe une mesure qui décroît à chaque étape pour une bonne stratégie de réécriture (des règles probabilistes), on peut appliquer notre résultat de convergence. Le choix de cette mesure qui décroît *à chaque étape* a été motivé par le fait que ce genre de critère est plus facile à vérifier sur les exemples. Quand on s'intéresse à des algorithmes équitables (*i.e.* où tout processus effectue infiniment souvent une action), on peut prouver des propriétés du genre “la mesure ϕ va décroître au bout d'un nombre fini d'étapes”, par exemple la prochaine fois que l'on réécrit un certain processus. Par contre, quand on ne suppose pas le système équitable (comme dans certains exemples de cette thèse), on n'a pas de raison simple pour s'assurer qu'en un nombre fini d'étapes (sans préciser lequel) une certaine mesure va décroître.
Lorsqu'on a montré qu'un système converge, il peut être intéressant de savoir à quelle “vitesse” on arrive dans l'ensemble souhaité. C'est pourquoi nous avons utilisé une méthode de regroupement d'états appelée *lumping* qui, en se basant sur la mesure calculée plus haut, permet dans certains cas de réduire le nombre de configurations à considérer, et ainsi de faciliter le calcul du temps moyen de convergence, ce qui a été appliqué sur des exemples.

Nous avons ensuite utilisé notre critère de convergence pour prouver la convergence d'une variante sans équité de l'algorithme du dîner des philosophes probabilistes, montrant ainsi que, si l'on enlève les transitions invariantes (*i.e.* qui ne font pas évoluer le système), l'hypothèse d'équité n'est pas nécessaire sur cet algorithme.
Cette variante et le fait de ne plus considérer des transitions invariantes nous ont permis de calculer une borne inférieure exponentielle sur le temps moyen de convergence évalué en nombre de transitions, pour un certain ordonnancement malicieux.
Si on compare ce résultat avec celui du cas d'un ordonnancement équitable qui annonce que le

temps moyen de convergence est un nombre constant de tours (*i.e.* intervalle de temps pendant lequel tout processus est sélectionné au moins une fois), on se rend bien compte que, même dans le cas équitable, un tour peut correspondre à un très grand nombre de transitions. En conclusion, on peut dire que si le temps d'un tour a une signification réelle et est borné (par exemple si un philosophe agit toutes les au plus 2 minutes), alors évaluer le temps moyen en tours a un sens. Par contre si c'est juste une façon de désigner un ensemble de transitions, alors cela ne donne pas une idée juste du temps que peut prendre la convergence.

Systèmes paramétrés déterministes

La deuxième approche que nous avons présentée concerne l'étude de systèmes distribués paramétrés en anneaux pour lesquels toutes les règles sont déterministes. Nous somme partis d'un travail réalisé dans l'article [BBFM01] de Beauquier, Bérard, Fribourg et Magniette, qui utilise des techniques de réécriture et un résultat de Dershowitz [Der81] permettant de réorganiser l'ordre des transitions le long d'une exécution, afin de prouver l'auto-stabilisation d'algorithmes sur des anneaux paramétrés.

Contrairement à la partie précédente, ici la topologie en anneau est importante car elle permet de représenter les configurations comme des mots et d'assimiler les transitions à des étapes de réécriture.

Notre travail est en fait une étape de plus vers l'automatisation de preuves de convergence. Plus précisément, nous avons prouvé que, sous certaines conditions sur les règles de réécriture, il suffisait de considérer des exécutions combinant réécriture préfixe et suffixe en parallèle afin de calculer un ensemble inévitable. En se basant sur un résultat dû à Caucal [Cau90], nous avons utilisé le fait que le langage engendré par réécriture préfixe et suffixe est régulier et peut être calculé en temps polynomial.

Nous avons en fait exhibé deux critères permettant de calculer un ensemble inévitable. Le premier est la fermeture de l'ensemble $\mathcal{N_S}^*$ engendré par réécriture préfixe et suffixe pour le système de réécriture $\mathcal{S}$. Comme cet ensemble $\mathcal{N_S}^*$ est régulier, la fermeture pour $\mathcal{S}$ est décidable et on peut vérifier automatiquement si le critère est satisfait. Le second critère est pour sa part vérifiable directement sur les règles, à savoir que l'on vérifie si le long des dérivations il peut y avoir ce que l'on appelle des *demi-tours*. Nous avons montré qu'en l'absence de tels demi-tours le système, que nous appelons alors *unidirectionnel* vérifie la fermeture de $\mathcal{N_S}^*$, et donc un ensemble appelé $Gr(\mathcal{N_S}^*)$ est inévitable.

Nous avons ensuite appliqué cette méthode de calcul d'ensemble inévitables à différents exemples, que ce soit afin de montrer l'auto-stabilisation d'algorithmes ([BD95, Gho93]), ou en adaptant la méthode afin de vérifier la propriété de vivacité d'un algorithme de détection de terminaison ([DFvG83]).

Perspectives

Dans la partie déterministe, nous avons obtenu un critère décidable (systèmes unidirectionnels) utilisé pour appliquer notre méthode, en obtenant que l'ensemble $Gr(\mathcal{N_S}^*)$ est inévitable. Une des pistes à envisager serait d'essayer de trouver un critère moins restrictif sur les systèmes de réécriture et permettant néanmoins de calculer un ensemble inévitable intéressant. Si par exemple, au lieu de montrer que $Gr(\mathcal{N_S}^*)$ est inévitable, on a que $Gr(\mathcal{N_S}^*)$ est inclus dans $\mathcal{L}$ et $\mathcal{R_S}^* \circ Gr(\mathcal{N_S}^*)$ est inévitable, alors si $\mathcal{L}$ est clos, $\mathcal{L}$ est absorbant. Un de nos objectifs est donc de trouver des critères décidables pour assurer ce genre de propriétés.

Dans le cadre probabiliste, notre méthode de preuve de convergence ne dépend pas de la topologie. C'est plutôt le genre de mesure que nous utilisons sur les exemples qui est adapté à la topologie et à la nature de l'algorithme. Ainsi il pourrait être intéressant d'utiliser en pratique ce genre de méthode sur des algorithmes fonctionnant sur des topologies autres que les anneaux, comme par exemple les arbres.

Bibliographie

[AK86] Krzysztof R. Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6) :307–309, 1986.

[APZ03] Tamarah Arons, Amir Pnueli, and Lenore D. Zuck. Parameterized verification by probabilistic abstraction. In *Proc. 6th Int. Conf. on Foundations of Software Science and Computational Structures (FOSSACS'03)*, volume 2620 of *LNCS*, pages 87–102. Springer, 2003.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *Theory of Parsing, Translation and Compiling*, volume I : Parsing. Prentice Hall, 1972.

[Bai98] Christel Baier. On the algorithmic verification of probabilistic systems. Habilitation thesis,Universität Mannheim, 1998.

[BBF+01] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification*. Springer, 2001.

[BBFM01] Joffroy Beauquier, Béatrice Bérard, Laurent Fribourg, and Frédéric Magniette. Proving convergence of self-stabilizing systems using first-order rewriting and regular languages. *Distributed Computing*, 14(2) :83–95, 2001.

[BC] Yves Bertot and Pierre Castéran. Le Coq' art. Electronic book available at `http ://www-sop.inria.fr/lemme/Yves.Bertot/coqart.html`.

[BCD95] Joffroy Beauquier, Stéphane Cordier, and Sylvie Delaët. Optimum probabilistic self-stabilization on uniform rings. In *Proc. 2nd Workshop on Self-Stabilizing Systems (WSS'95)*, 1995.

[BD94] Joffroy Beauquier and Sylvie Delaët. Probabilistic self-stabilizing mutual exclusion in uniform rings. In *Proc. 13th Annual ACM Symposium on Principles of Distributed Computing (PODC'94)*, page 378. ACM Press, 1994.

[BD95] Joffroy Beauquier and Olivier Debas. An optimal self-stabilizing algorithm for mutual exclusion on uniform bidirectional rings. In *Proc. 2nd Workshop on Self-Stabilizing Systems (WSS'95)*, pages 226–239, 1995.

[BdA95] Andrea Bianco and Luca de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proc. 15th Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'95)*, volume 1026 of *LNCS*, pages 499–513. Springer, 1995.

[BDLGJ02] Joffroy Beauquier, Jérôme Durand-Lose, Maria Gradinariu, and Colette Johnen. Token based self-stabilizing uniform algorithms. *Journal of Parallel and Distributed Computing*, 62(5) :899–921, 2002.

[BF99] Béatrice Bérard and Laurent Fribourg. Reachability analysis of (timed) Petri nets using real arithmetic. In *Proc. 10th Int. Conf. on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 178–193. Springer, 1999.

[BGJ99a] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Memory space requirements for self-stabilizing leader election protocols. In *Proc. 18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*, pages 199–208. ACM Press, 1999.

[BGJ99b] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. Technical Report 1225, L.R.I., Orsay, France, Sept. 1999.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model-checking. In *Proc. 12th Conf. on Computer-Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 403–418. Springer, 2000.

[BM01] Thomas Bonald and Laurent Massoulié. Impact of fairness on internet performance. In *Proc. of Joint Int. Conf. on Measurements and Modeling of Computer Systems (SIGMETRICS/Performance 2001)*, pages 82–91, 2001.

[BMT01] Ahmed Bouajjani, Anca Muscholl, and Tayssir Touili. Permutation rewriting and algorithmic verification. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, pages 399–408. IEEE publishing, 2001.

[BO93] Ronald V. Book and Friedrich Otto. *String-Rewriting Systems*. Springer, 1993.

[Bré99] Pierre Brémaud. *Markov chains - Gibbs fields, Monte Carlo simulation, and queues*. Springer, 1999.

[BS03] Nathalie Bertrand and Philippe Schnoebelen. Model checking lossy channels systems is probably decidable. In *Proc. 6th Int. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'03)*, volume 2620 of *LNCS*, pages 120–135. Springer, 2003.

[BT01] Ed Brinksma and Jan Tretmans. Testing transition systems : An annotated bibliography. In *Proc. 4th Summer school on Modeling and Verification of Parallel Processes (MOVEP'00)*, volume 2067 of *LNCS*, pages 187–195. Springer, 2001.

[Büc89] J. Richard Büchi. *Finite automata, their algebras and grammars - Towards a theory of formal expressions*. Springer, 1989. Dirk Siefkes Ed.

[Cau88] Didier Caucal. Récritures suffixes de mots. Technical Report 871, INRIA, Rennes, France, Jul. 1988.

[Cau90] Didier Caucal. On the regular structure of prefix rewriting. In *Proc. 15th Coll. Trees in Algebra and Programming (CAAP'90)*, volume 431 of *LNCS*, pages 87–102. Springer, 1990.

[CGJ95] Edmund M. Clarke, Orna Grumberg, and Somesh Jha. Veryfying parameterized networks using abstraction and regular languages. In *Proc. 6th Int. Conf. on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 395–407. Springer, 1995.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[CLC03] Hubert Comon-Lundh and Véronique Cortier. Security properties : two agents are sufficient. In *Proc. 12th European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 99–113. Springer, 2003.

[CO98] Jon Crowcroft and Philippe Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing TCP. *ACM Computer Communication Review*, 28(3) :53–69, 1998.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. *available at* `http ://www.csl.sri.com/papers/wift-tutorial/`, 1995.

[dA97] Luca de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, Dec. 1997.

[dA99] Luca de Alfaro. Computing minimum and maximum reachability times in probabilistic systems. In *Proc. 10th Int. Conf. on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 66–81. Springer, 1999.

[Del95] Sylvie Delaët. *Auto-Stabilisation : Modèle et applications à l'exclusion mutuelle*. PhD thesis, Université de Paris Sud, Dec. 1995.

[Der81] Nachum Dershowitz. Termination of linear rewriting systems. In *Proc. 8th Int. Coll. Automata, Languages and Programming (ICALP'81)*, volume 115 of *LNCS*, pages 448–458. Springer, 1981.

[DFN01] Marie Duflot, Laurent Fribourg, and Ulf Nilsson. Unavoidable configurations of parameterized rings of processes. In *Proc. 12th Int. Conf. on Concurrency Theory (CONCUR'01)*, volume 2154 of *LNCS*, pages 472–486. Springer, 2001.

[DFP01] Marie Duflot, Laurent Fribourg, and Claudine Picaronny. Randomized finite-state distributed algorithms as Markov chains. In *Proc. 15th Int. Conf. on Distributed Computing (DISC'01)*, volume 2180 of *LNCS*, pages 240–254. Springer, 2001.

[DFP02] Marie Duflot, Laurent Fribourg, and Claudine Picaronny. Randomized dining philosophers without fairness assumption. In *Proc. 2nd IFIP Int. Conf. on Theoretical Computer Science (TCS@02)*, volume 223 of *IFIP Conference Proceedings*, pages 169–180. Kluwer Academic, 2002. Long version accepted for publication in the journal *Distributed Computing*.

[DFvG83] Edsger W. Dijkstra, W.H.J Feijen, and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16 :217–219, 1983.

[Dij72] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. In *Operating Systems Techniques*, pages 72–93. Academic Press, 1972.

[Dij73] Edsger W. Dijkstra. Ewd 391 : Self-stabilization in spite of distributed control. In *Selected Writings on Computing : A personnal perspective*, pages 41–46. Springer-Verlag, 1973. (written in 1973, published in 1982).

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11) :643–644, 1974.

[DIM95] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Analyzing expected time by scheduler-luck games. *IEEE Transactions on Softare Engineering*, 21(5) :429–439, 1995.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science - Formal Models and Semantics*, volume B, chapter 6, pages 243–320. Elsevier, 1990.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[Eme90] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science - Formal Models and Semantics*, volume B, chapter 16, pages 995–1072. Elsevier, 1990.

[EN95] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 85–94. ACM Press, 1995.

[FD94] Mitchell Flatebo and Ajoy Kumar Datta. Two-state self-stabilizing algorithms for token rings. *IEEE Transactions on Software Engineering*, 20(6) :500–504, 1994.

[FL02] Alain Finkel and Jérôme Leroux. How to compose presburger-accelerations : Applications to broadcast protocols. In *Proc. 22nd Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'02)*, volume 2556 of *LNCS*, pages 145–156. Springer, 2002.

[Fle02] Emmanuel Fleury. *Les automates temporisés avec mises à jour*. PhD thesis, École Normale Supérieure de Cachan., Dec. 2002.

[FO97] Laurent Fribourg and Hans Olsén. Reachability sets of parametrized rings as regular languages. In *Proc. 2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97)*, volume 9 of *ENTCS*. Elsevier Science, 1997.

[Gho93] Sukumar Ghosh. An alternative solution to a problem on self-stabilization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4) :735–742, 1993.

[GRK99] Panos Gevros, Fulvio Risso, and Peter Kirstein. Analysis of a method for differential TCP service. In *Proc. 4th Symposium on Global Internet (GLOBECOM'99)*, 1999.

[Her90] Ted Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2) :63–67, 1990.

[Her02] Oltea Mihaela Herescu. *The probabilistic asynchronous Pi-calculus*. PhD thesis, Department of Computer Science and Engineering, Pennsylvania State university, Dec. 2002.

[Hoe94] Jaap-Henk Hoepman. Uniform deterministic self-stabilizing ring-orientation on odd-length rings. In *Proc. 8th Workshop on Distributed Algorithms (WDAG'94)*, volume 857 of *LNCS*, pages 265–279. Springer, 1994.

[HP00] Oltea Mihaela Herescu and Catuscia Palamidessi. Probabilistic asynchronous π-calculus. In *Proc. 3rd Int. Conf. on Foundations of Software, Science and Computation Structures (FoSSaCS'00)*, volume 1784 of *LNCS*, pages 146–160. Springer, 2000.

[HP01] Oltea Mihaela Herescu and Catuscia Palamidessi. On the generalized dining philosophers problem. In *Proc. 20th Annual ACM Symposium on Principles of Distributed Computing (PODC'01)*, pages 81–89. ACM Press, 2001.

[IJ90] Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proc. 9th Annual ACM Symposium on Principles of Distributed Computing (PODC'90)*, pages 119–131. ACM Press, 1990.

[IJ93] Amos Israeli and Marc Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 104(2) :175–196, 1993.

[JN00] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Proc. 6th Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 220–234. Springer, 2000.

[KM95] Robert P. Kurshan and Kenneth L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117(1) :1–11, 1995.

[KMM+97] Yonit Kesten, Oded Maler, Monica Marcus, Amir Pnueli, and Elad Shahar. Symbolic model-checking with rich assertional languages. In *Proc. 9th Conf. on Computer-Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 424–435. Springer, 1997.

[KNP02] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM : Probabilistic symbolic model checker. In *Proc. 12th Int. Conf. on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.

[KPSZ02] Yonit Kesten, Amir Pnueli, Elad Shahar, and Lenore D. Zuck. Network invariants in action. In *Proc. 13th Conf. on Concurency Theory (CONCUR'02)*, volume 2421 of *LNCS*, pages 101–115. Springer, 2002.

[KS60] John G. Kemeny and J. Laurie Snell. *Finite Markov Chains*. D. van Nostrand Co., 1960.

[KSK66] John G. Kemeny, J. Laurie Snell, and Anthony W. Knapp. *Denumerable Markov Chains*. D. van Nostrand Co., 1966.

[KY97] Hirotsugu Kakugawa and Masafumi Yamashita. Uniform and self-stabilizing token rings allowing unfair daemon. *IEEE Trans. Parallel and Distributed Systems*, 8(2) :154–163, 1997.

[LLM+02] Sophie Laplante, Richard Lassaigne, Frédéric Magniez, Sylvain Peyronnet, and Michel de Rougemont. Probabilistic abstraction for model checking : An approach based on property testing. In *Proc. of 17th IEEE Symposium on Logic In Computer Science (LICS'02)*, volume 8, pages 30–39. IEEE publishing, 2002.

[LR81] Daniel J. Lehmann and Michael O. Rabin. The advantages of free choice : a symmetric and fully-distributed solution to the dining philosophers problem. In *Proc. 8th Annual ACM Symposium on Principles of Programming Languages(POPL'81)*, pages 133–138. ACM Press, 1981.

[LS91] Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94 :1–28, 1991.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3) :382–401, 1982.

[LSS94] Nancy A. Lynch, Isaac Saias, and Roberto Segala. Proving time bounds for randomized distributed algorithms. In *Proc. 13th Annual ACM Symposium on Principles of Distributed Computing (PODC'94)*, pages 314–323. ACM Press, 1994.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[Mag02] Frédéric Magniette. *Preuves d'algorithmes auto-stabilisants*. PhD thesis, Université Paris Sud, Jun. 2002.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - Specification*. Springer, 1992.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus for mobile processes, I and II. *Information and Computation*, 100(1) :1–40 & 41–77, 1992.

[Nor98] James R. Norris. *Markov Chains*. Cambridge University press, 1998.

[NP96] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. In *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 179–194. Springer, 1996. A long version appeared in the Journal *Information and Computation*, 163(1) :1–59, 2000.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Pal97] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 256–265. ACM Press, 1997.

[Pan01] Prakash Panangaden. Measure and probability for concurrency theorists. *Theoretical Computer Science*, 253(2) :287–309, 2001.

[PH02] Catuscia Palamidessi and Oltea Mihaela Herescu. A randomized distributed encoding of the pi-calculus with mixed choice. In *Proc. 2nd IFIP Int. Conf. on Theoretical Computer Science (TCS@02)*, volume 223 of *IFIP Conference Proceedings*, pages 537–549. Kluwer Academic, 2002.

[PS95] Anna Pogosyants and Roberto Segala. Formal verification of timed properties for randomized distributed algorithms. In *Proc. 14th Annual ACM Symposium on Principles of Distributed Computing (PODC'95)*, pages 174–183. ACM Press, 1995.

[PS00] Amir Pnueli and Elad Shahar. Liveness and acceleration in parameterized verification. In *Proc. 12th Conf. on Computer-Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 328–343. Springer, 2000.

[Put94] Martin L. Puterman. *Markov Decision Processes : Discrete Stochastic Dynamic Programming*. Wiley-Interscience, 1994.

[PZ86] Amir Pnueli and Lenore D. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1) :53–72, 1986.

[Rab63] Michael O. Rabin. Probabilistic automata. *Information and control*, 6 :230–245, 1963.

[RL94] Michael O. Rabin and Daniel J. Lehmann. *The advantages of free choice : a symetric and fully distributed solution for the dining philosophers problem. In "A Classical Mind : Essays in Honour of C.A.R. Hoare"*, chapter 20, pages 333–352. Prentice Hall, 1994.

[Rus01] John Rushby. Theorem proving for verification. In *Proc. 4th Summer school on Modeling and Verification of Parallel Processes (MOVEP'00)*, volume 2067 of *LNCS*, pages 39–57. Springer, 2001.

[SBB+99] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels - Techniques et outils du model-checking.* Vuibert, 1999.

[Sch93] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1) :45–67, 1993.

[Seg95] Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Massachusetts Institue of Technology, Jun. 1995.

[SL95] Roberto Segala and Nancy A. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2) :250–273, 1995.

[Tel91] Gerard Tel. *Topics in Distributed Algorithms.* Cambridge University Press, 1991.

[Tel94] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[Tou01] Tayssir Touili. Regular model checking using widening techniques. In *Proc. 1st Workshop on Verification of Parameterized Systems (VEPAS'01)*, volume 50(4) of *ENTCS*. Elsevier, 2001.

[Val01] Antti Valmari. Composition and abstraction. In *Proc. 4th Summer school on Modeling and Verification of Parallel Processes (MOVEP'00)*, volume 2067 of *LNCS*, pages 58–98. Springer, 2001.

[Var85] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proc. 26th Annual Symp. on Foundations of Computer Science(FOCS'85)*, pages 327–338. IEEE Comp. Soc. Press, 1985.

[vGSS95] Rob J. van Glabbeek, Scott A. Smolka, and Bernhard Steffen. Reactive, generative and stratified models of probabilistic processes. *Information and Computation*, 121(1) :59–80, 1995.

[vN00] Gertjan van Noord. FSA6 reference manual, 2000. Available on `http ://odur.let.rug.nl/~vannoord/Fsa/`.

[ZPK02] Lenore D. Zuck, Amir Pnueli, and Yonit Kesten. Automatic verification of probabilistic free choice. In *Proc. 3rd Int. Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'02)*, volume 2294 of *LNCS*, pages 208–224. Springer, 2002.

Index

π-calcul, 112
σ-algèbre, 26
  engendrée, 26

absorbant
  chaîne de Markov, 32
  ensemble, 58
  état, 32
acceptable
  configuration, 90
action, 40
activable
  position, 50
active
  partie, 127
adversaire, 52
algorithme
  symétrique, 88
  totalement distribué, 88
anneau, 49
anti-jeton, 101
  coefficient, 103
  orienté, 102
  poids, 103
arbre des exécutions, 37
  lumpé, 82
asynchrone
  communication, 46
auto-stabilisant
  système, 57
  vers un ensemble, 123
auto-stabilisation, 57

bidirectionnel, 47

centralisé
  démon, 53
  ordonnancement, 53
chaîne de Markov, 29
  absorbante, 32
  homogène, 29
chaîne top, 126
  généralisée, 130
  quasi cyclique, 126
chemin, 30
  fini, 30
clos
  mot, 58, 120
  schéma, 129
close
  dérivation, 122
  instance, 120
  réduction, 121
  substitution, 120
  surréduction, 142
coefficient
  d'anti-jeton, 103
  de jeton, 100
communication
  asynchrone, 46
  synchrone, 46
compassion, 53
composante fortement connexe, 32
configuration, 40
  acceptable, 90
convergence, 56
  probabiliste, 33
  temps moyen, 33
cyclique
  dérivation, 122
cylindre, 30

découpage
  d'un mot, 136
démon, 42, 52

centralisé, 53
distribué, 53
équitable, 54
synchrone, 53
dérivation
close, 122
cyclique, 122
déterministe
ordonnancement, 36
disjoints
jetons, 99
distribué
démon, 53
système, 39
totalement, 88
distribution, 27

ensemble
absorbant, 58
des observables, 26
fermé, 58
inévitable, 57
initial, 140
légitime, 57
équitable
algorithme, 54
démon, 54
faiblement, 53
fortement, 53
ordonnancement, 54
système, 122
espérance, 34
état
absorbant, 32
d'une chaîne de Markov, 29
global, 40
local, 40
récurrent, 31
transitoire, 31
étiquette
d'une transition, 136
événement, 26
exécution, 35, 51
arbre des, 37
finie, 35, 51
extension
préfixe, 139
suffixe, 139

facteur, 50
faiblement équitable, 53
fermé
ensemble, 58
fortement équitable, 53

généralisée
réduction, 129

homogène
chaîne de Markov, 29

inactive
partie, 127
indépendant
événement, 28
indice, 50
d'un anti-jeton, 101
d'un jeton, 99
inévitable
ensemble, 57
instance, 120
close, 120

jeton, 43
coefficient, 100
poids, 100
pour le dîner des philosophes, 99
justice, 53

légitime
ensemble, 57
linéaire, 49
liste
d'anti-jetons, 101
de jetons, 100
lumpable
système, 81

markovien
ordonnancement, 36
processus de décision, 34
système, 29
matrice
de transition, 29
stochastique, 29
membre

- droit, 50
- gauche, 50

mesurable
- ensemble de chemins, 31
- espace, 27
- fonction, 27

mot, 49
- clos, 120
- du premier ordre, 120

motif, 50

nœtherien
- système, 122

nœud, 40

observables
- ensemble des, 26

ordonnancement, 35, 36, 52
- centralisé, 53
- déterministe, 36
- équitable, 54
- markovien, 36
- sans mémoire, 36
- synchrone, 53

ordonnanceur, 52

panne
- byzantine, 45
- définitive, 45
- transitoire, 45

partie
- active, 127
- inactive, 127

poids
- d'un anti-jeton, 103
- d'un jeton, 100

position, 50
- activable, 50

préfixe
- extension, 139
- réduction, 135

premier ordre
- mot du, 120
- schéma du, 129

privilège, 133

probabilité, 26
- conditionnelle, 28
- espace de, 27
- maximale, 37
- mesure de, 27
- minimale, 37

processus de décision markovien, 34

progrès, 88

quasi-cyclique
- chaîne top, 126

read
- all, 46
- one, 46

récurrent
- état, 31

réductible
- de façon minimale, 125

réduction
- close, 121
- dans un schéma, 129
- généralisée, 129
- préfixe, 135
- suffixe, 135

réécriture
- préfixe, suffixe, 135
- règle de, 50
- stratégie de, 69
- système de, 50

règle
- de réécriture, 50

sans mémoire
- ordonnancement, 36

schéma, 128
- clos, 129
- du premier ordre, 129

stratégie
- de réécriture, 69

substitution, 120
- close, 120
- minimale, 125

suffixe
- extension, 139
- réduction, 135

surréduction, 125
- close, 142

symétrique
- algorithme, 88

synchrone

communication, 46
démon, 53
ordonnancement, 53
système
auto-stabilisant, 57
vers un ensemble, 123
bidirectionnel, 47
de réécriture, 50
de transitions probabiliste, 30
distribué, 39
en anneau, 49
équitable, 122
linéaire, 49
lumpable, 81
markovien, 29
nœtherien, 122
unidirectionnel, 47, 146

temps moyen, 33
topologie
en anneau, 47
en arbre, 47
en étoile, 47
linéaire, 49
tour, 91
transition, 40
transitoire
état, 31
panne, 45
traverser
un ensemble, 70
tribu, 26

unidirectionnel
système de réécriture, 146
système distribué, 47
unificateur, 124

variable
aléatoire, 28
du premier ordre, 120
voisin, 47